#1 En complementos de aprendizaje: Presentaciones - Videos - Ejercicios

Aprenda Practicando

2ª Edición

Introducción a la Programación

Algoritmos y su implementación en

VB.NET, C#, Java y C++

Felipe Ramírez

- Aprenda los elementos y técnicas de la lógica de programación
- Aprenda el proceso de análisis, abstracción y documentación de casos reales de negocios
- Desarrolle algoritmos, diagramas de flujo, miniespecificaciones y pruebas de escritorio
- Implemente algoritmos en Visual Basic.NET, C#, C++ Java y Raptor
- Aprenda los elementos de la programación orientada a objetos, usando Visual Basic

Alfaomega

Introducción a la Programación

Algoritmos y su implementación en VB.NET, C#, Java y C++

www.detodoprogramacion.com

Introducción a la Programación

Algoritmos y su implementación en VB.NET, C#, Java y C++

2ª Edición

Felipe Ramírez

Master en Informática Administrativa,
Licenciado en Informática Administrativa,
ITIL Certified Trainer,
Licenciado en Derecho y Ciencias Sociales,
Catedrático de la Facultad
de Contaduría Pública y Administración,
Universidad Autónoma de Nuevo León, México.

Revisión Técnica:

Lic. Álvaro Francisco Salazar González
Facultad de Contaduría Pública y Administración,
Universidad Autónoma de Nuevo León, México.

Alfaomega

www.detodoprogramacion.com

Datos catalográficos
Ramírez, Felipe
Introducción a la Programación. Algoritmos y su implementación en Visual Basic.NET, C#, Java y C++
Segunda Edición
Alfaomega Grupo Editor, S.A. de C.V., México
ISBN: 978-970-15-1280-7
Formato: 17 x 23 cm Páginas: 520

Introducción a la Programación.
Algoritmos y su implementación en Visual Basic.NET, C#, Java y C++
Felipe Ramírez

Segunda edición: Alfaomega Grupo Editor, junio de 2007

© 2007 ALFAOMEGA GRUPO EDITOR, S.A. de C.V.
Pitágoras 1139, Col. Del Valle, 03100, México D.F.

Miembro de la Cámara Nacional de la Industria Editorial Mexicana
Registro No. 2317

Pág. Web: **http://www.alfaomega.com.mx**
E-mail: **libreriapitagoras@alfaomega.com.mx**

ISBN: 978-970-15-1280-7

Derechos reservados.
Esta obra es propiedad intelectual de su autor y los derechos de publicación en lengua española han sido legalmente transferidos al editor. Prohibida su reproducción parcial o total por cualquier medio sin permiso por escrito del propietario de los derechos del copyright.

Edición autorizada para venta en México y todo el continente americano.

Impreso en México. Printed in Mexico.

Empresas del grupo:

México: Alfaomega Grupo Editor, S.A. de C.V. – Pitágoras 1139, Col. Del Valle, México, D.F. – C.P. 03100.
Tel.: (52-55) 5089-7740 – Fax: (52-55) 5575-2420 / 2490. Sin costo: 01-800-020-4396
E-mail: libreriapitagoras@alfaomega.com.mx

Colombia: Alfaomega Colombiana, S.A. – Carrera 15 No. 64 A 29 – PBX (57-1) 2100122
Fax: (57-1) 6068648 – E-mail: scliente@alfaomega.com.co

Chile: Alfaomega Grupo Editor, S.A. – Dr. Manuel Barros Borgoño 21 Providencia, Santiago, Chile
Tel.: (56-2) 235-4248 – Fax: (56-2) 235-5786 – E-mail: agechile@alfaomega.cl

Argentina: Alfaomega Grupo Editor Argentino, S.A. – Paraguay 1307 P.B. "11", Capital Federal, Buenos Aires, C.P. 1057 – Tel.: (54-11) 4811-7183 / 8352, E-mail: agea@fibertel.com.ar

www.detodoprogramacion.com

En mi casa me espera
una rosa y un rosal.

www.detodoprogramacion.com

Índice de contenido

Índice de contenido i
Indice de prácticas ix
Prólogo xi
Cómo utilizar este libro xv

Capítulo 1: La computadora y los niveles de datos

Computadora 3
Tipos de dispositivos 4
Programas 5
Niveles de datos 6
En términos físicos 6
En términos de relevancia 8
Estimando el tiempo de descarga de una película 9
Identificando los niveles de utilidad de los datos 9
Identificando un entorno operativo 10
Manipulación de datos 11
Mapa mental del capítulo 13
Terminología 14
Preguntas 14
Examen rápido 15

Capítulo 2: Lenguajes de programación y su clasificación

Lenguajes de programación 20
Tipos de código 23
Construyendo secuencias binarias ejecutables 24
Compiladores 25
Utilidad del código intermedio 25
Clasificaciones de los lenguajes de programación 26
Clasificación de los lenguajes en cuanto a su generación 26
Clasificación de los lenguajes en cuanto a su nivel 28
Clasificación de los lenguajes en cuanto a su propósito 28
Clasificación de los lenguajes en cuanto a su orientación 29
Mapa mental del capítulo 31
Terminología 32
Preguntas 32
Examen rápido 33

Capítulo 3: Introducción a la lógica de programación

Lógica 37
Silogismos, proposiciones y premisas 37
Calidad de las premisas y las conclusiones 39
Reconociendo la calidad de premisas y conclusiones 41
Lógica de programación 42
Aplicación de la lógica en la programación 43

www.detodoprogramacion.com

Finalidad de los enunciados en el proceso 44
Principios en la elaboración de enunciados 46
Principios relacionados con los datos de entrada 46
Principios relacionados con el proceso 47
Principios relacionados con los datos de salida 47
Analizando un silogismo y clasificando las premisas 48
Mapa mental del capítulo 50
Terminología 51
Preguntas 51
Examen rápido 53

Capítulo 4: La naturaleza de los datos

Naturaleza de los valores 57
Propiedades de los datos 58
Expresiones y variables 59
Expresiones 59
Variables 59
Arreglos 60
Análisis de los nombres de variables 61
Los tipos de datos y su soporte en los lenguajes 62
Visual Basic.NET 62
C# 63
C++ 63
Java 64
Tipos de datos base 64
Dominios 65
Dominio de tipo 65
Dominio de regla de negocio 65
Dominio de relación 65
Representación abstracta de tipos base 66
Tipos de datos 66
Dominios 66
Identificando los tipos de datos idóneos para los datos 68
Análisis de economía de datos 69
Representación abstracta de dominios 70
Mapa mental del capítulo 71
Terminología 72
Preguntas 72
Examen rápido 73

Capítulo 5: Operadores y reglas de precedencia

Categorías de operadores 76
Operadores aritméticos 76
Operadores de asignación 78
Operadores comparativos 78
Operadores lógicos 79
Reglas de precedencia 80
Precedencia implícita 81
Precedencia posicional 82
Precedencia explícita 82
Aplicación de operadores y sus reglas de precedencia 83

Representación de expresiones complejas operadores y elementos de lógica simbólica... 85
Mapa mental del capítulo 88
Terminología 89
Preguntas 89
Examen rápido 91

Capítulo 6: Algoritmos para el análisis de casos reales

Procedimiento de creación de un programa 95
Ciclo de desarrollo 95
Relevancia de las fases 97
Implementación práctica del ciclo de desarrollo 98
Exposición de casos prácticos 100
Caso 1: Mensaje 101
Caso 2: Niveles de servicio 101
Caso 3: Muestra de una tabla de multiplicar 102
Caso 4: Muestra las tablas de multiplicar del 1 al 5 102
Caso 5: Análisis de promedios 102
Analizar el caso real 102
Determinar de forma clara los requerimientos del cliente 103
Determinar el alcance del programa 106
Determinar la interfaz y el comportamiento de un sistema 108
Algoritmos 109
Analizando los casos y diseñando sus algoritmos 110
Caso 1: Mensaje 110
Caso 2: Niveles de servicio 111
Caso 3: Muestra de una tabla de multiplicar 114
Caso 4: Muestra las tablas de multiplicar del 1 al 5 116
Caso 5: Análisis de promedios 118
Comprobando habilidades en el análisis de casos y el diseño de algoritmos 120
Mapa mental del capítulo 130
Terminología 131
Preguntas 131
Examen rápido 133

Capítulo 7: Algoritmos orientados a datos y miniespecificaciones

Algoritmos orientados a los datos 136
Transformando algoritmos a su versión orientada a datos 138
Operaciones a nivel dato 139
Declaración 140
Cambio de estado 140
Muestra de datos 142
Estructuras de decisión y control 143
Estructuras de decisión 144
Contadores y acumuladores 145
Estructuras de control 146
Anidamiento 149
Arreglos 150
Miniespecificaciones 152
Cómo elaborar una miniespecificación 153
Miniespecificación de los casos prácticos 154
Caso 1: Mensaje 155
Caso 2: Niveles de servicio 156
Caso 3: Muestra de una tabla de multiplicar 157

Caso 4: Muestra las tablas de multiplicar del 1 al 5 159
Caso 5: Análisis de promedios 160
Pruebas de escritorio 163
Comprobando habilidades en el análisis de casos y el diseño de algoritmos orientados a datos 165
Mapa mental del capítulo 171
Terminología 172
Preguntas 172
Examen rápido 173

Capítulo 8: Diagramas de flujo

Diagramas de flujo 177
Ventajas de utilizar diagramas de flujo 177
Estándar ANSI/ISO 5807-1985 para diagramas de flujo 178
Símbolos utilizados en los diagramas de flujo 178
Reglas para la elaboración de diagramas de flujo 181
Cómo elaborar un diagrama de flujo de programa 182
Diagramas de flujo de los casos prácticos 187
Caso 1: Mensaje 188
Caso 2: Niveles de servicio 189
Caso 3: Muestra de una tabla de multiplicar 191
Caso 4: Muestra las tablas de multiplicar del 1 al 5 193
Caso 5: Análisis de promedios 195
Comprobando habilidades en el diseño de diagramas de flujo 198
Mapa mental del capítulo 204
Terminología 205
Preguntas 205
Examen rápido 207

Capítulo 9: Programación visual usando Raptor

Raptor como herramienta de desarrollo 210
Capacidades del ambiente de desarrollo 211
Ventana principal (main) 213
Consola maestra (master console) 215
Ejecución de comandos desde master console 216
Instrucciones soportadas 218
Procedimiento de creación de un programa 219
Integrando instrucciones a un programa 222
Declaración de variables y arreglos, y uso de la instrucción assignment 227
Manejo de bucles usando Loop 230
Formas de ejecución de un programa en Raptor y el uso de pruebas de escritorio automáticas 233
Integrando peticiones de datos 235
Integrando condicionales y mostrado de datos 239
Manejo de puntos de interrupción (breakpoints) y comentarios 243
Creación y consumo de subcharts 246
Programación de casos práctic os usando Raptor 251
Caso 1: Mensaje 251
Caso 2: Niveles de servicio 252
Caso 3: Muestra de una tabla de multiplicar 253
Caso 4: Muestra las tablas de multiplicar del 1 al 5 255
Caso 5: Análisis de promedios 257

Desarrollando un programa de regla de tres simple usando Raptor *259*
Desarrollando un programa para el cálculo del área de un polígono irregular usando Raptor *259*
Desarrollando un programa con bucles, condicionales y arreglos usando Raptor *261*
Mapa mental del capítulo 262
Terminología 263
Preguntas 263
Examen rápido 265

Capítulo 10: Implementación de algoritmos en Visual Basic.NET

Elementos de un ambiente de desarrollo 269
Implementación de algoritmos en Visual Basic.NET 270
Ambiente de desarrollo 270
Estructura básica de un programa 273
Características generales del lenguaje 273
Tipos de datos 274
Declaración de variables y arreglos 274
Operadores principales 275
Cambios de estado 276
Mostrado de datos 277
Peticiones de datos 278
Estructuras de decisión 279
Estructuras de control 280
Compilación y ejecución de programas 282
Codificación de C1Mensaje en Visual Basic.NET *282*
Codificación de C2Niveles en Visual Basic.NET *284*
Codificación de C3Tabla en Visual Basic.NET *286*
Codificación de C4MultiTabla en Visual Basic.NET *288*
Codificación de C5Promedios en Visual Basic.NET *291*
Desarrollando un programa de regla de tres simple usando Visual Basic.NET *294*
Desarrollando un programa para el cálculo del área de un polígono irregular usando Visual Basic.NET *294*
Desarrollando un programa con bucles, condicionales y arreglos usando Visual Basic.NET *296*
Mapa mental del capítulo 297
Terminología 298
Preguntas 298
Examen rápido 299

Capítulo 11: Implementación de algoritmos en C#

Implementación de algoritmos en C# 302
Ambiente de desarrollo 302
Estructura básica de un programa 305
Características generales del lenguaje 305
Tipos de datos 306
Declaración de variables y arreglos 306
Operadores principales 307
Cambios de estado 308
Mostrado de datos 309
Peticiones de datos 309
Estructuras de decisión 311

Estructuras de control ... 312
Compilación y ejecución de programas ... 315
Codificación de C1Mensaje en C# ... 315
Codificación de C2Niveles en C# ... 317
Codificación de C3Tabla en C# ... 319
Codificación de C4MultiTabla en C# ... 321
Codificación de C5Promedios en C# ... 324
Desarrollando un programa de regla de tres simple usando C# ... 327
Desarrollando un programa para el cálculo del área de un polígono irregular usando C# ... 328
Desarrollando un programa con bucles, condicionales y arreglos usando C# ... 329
Mapa mental del capítulo ... 330
Terminología ... 331
Preguntas ... 331
Examen rápido ... 333

Capítulo 12: Implementación de algoritmos en C++

Implementación de algoritmos en C++ ... 336
Ambiente de desarrollo ... 336
Estructura básica de un programa ... 339
Características generales del lenguaje ... 339
Tipos de datos ... 340
Declaración de variables y arreglos ... 340
Operadores principales ... 341
Cambios de estado ... 342
Mostrado de datos ... 342
Peticiones de datos ... 343
Estructuras de decisión ... 344
Estructuras de control ... 346
Compilación y ejecución de programas ... 348
Codificación de C1Mensaje en C++ ... 349
Codificación de C2Niveles en C++ ... 350
Codificación de C3Tabla en C++ ... 352
Codificación de C4MultiTabla en C++ ... 355
Codificación de C5Promedios en C++ ... 357
Desarrollando un programa de regla de tres simple usando C++ ... 360
Desarrollando un programa para el cálculo del área de un polígono irregular usando C++ ... 361
Desarrollando un programa con bucles, condicionales y arreglos usando C++ ... 362
Mapa mental del capítulo ... 363
Terminología ... 364
Preguntas ... 364
Examen rápido ... 365

Capítulo 13: Implementación de algoritmos en Java

Implementación de algoritmos en Java ... 368
Ambiente de desarrollo ... 368
Estructura básica de un programa ... 371
Características generales del lenguaje ... 371
Tipos de datos ... 372
Declaración de variables y arreglos ... 372
Operadores principales ... 373
Cambios de estado ... 374
Mostrado de datos ... 375

Peticiones de datos ... 375
Estructuras de decisión ... 377
Estructuras de control ... 379
Compilación y ejecución de programas ... 381
Codificación de C1Mensaje en Java ... 382
Codificación de C2Niveles en Java ... 383
Codificación de C3Tabla en Java ... 385
Codificación de C4MultiTabla en Java ... 388
Codificación de C5Promedios en Java ... 390
Desarrollando un programa de regla de tres simple usando Java ... 394
Desarrollando un programa para el cálculo del área de un polígono irregular usando Java ... 394
Desarrollando un programa con bucles, condicionales y arreglos usando Java ... 396
Mapa mental del capítulo ... 397
Terminología ... 398
Preguntas ... 398
Examen rápido ... 399

Capítulo 14: Fundamentos de la programación orientada a objetos

Fundamentos de la programación orientada a objetos ... 401
Términos básicos de POO ... 403
Clases y objetos ... 403
Propiedades, métodos y eventos ... 403
Encapsulamiento (encapsulation) ... 404
Herencia (inheritance) ... 405
Polimorfismo (polymorphism) ... 406
Overloading, Overriding, y Shadowing ... 406
Elementos esenciales de los objetos que permiten su programación ... 407
Identidad ... 407
Estado ... 408
Comportamiento ... 409
Definición de una clase ... 410
Definición de propiedades ... 411
Creando una clase y generando una instancia ... 413
Definiendo y utilizando propiedades ... 414
Herencia ... 417
¿Cuándo se debe utilizar la herencia? ... 417
Inherits (Heredar) ... 418
NotInheritable (No Heredable) ... 418
MustInherit (Debe Heredar) ... 418
Modificadores de acceso, necesarios para la herencia ... 419
Estatutos auxiliares de herencia ... 419
NotOverridable (No Remplazable) ... 420
Overridable (Remplazable) ... 420
MustOverride (Debe Remplazarse) ... 420
Overrides (Remplaza) ... 421
Overloads (Sobrecarga/Sustituye) ... 421
MyBase y MyClass ... 421
Polimorfismo ... 422
Creando una jerarquía de clases por herencia ... 422
Comprobando el polimorfismo basado en herencia, y el uso de MyBase ... 424
Mapa mental del capítulo ... 429
Terminología ... 430
Preguntas ... 430

www.detodoprogramacion.com

Examen rápido del módulo 431

Anexo 1: Casos resueltos

Caso 1: Mensaje 434
Caso 2: Niveles de servicio 439
Caso 3: Muestra de una tabla de multiplicar 448
Caso 4: Muestra las tablas de multiplicar del 1 al 5 456
Caso 5: Análisis de promedios 464

Anexo 2: Tablas de referencia

Tipos de datos base en VB.NET, C#, C++ y Java 476
Comparativo de operadores VB.NET, C#, C++ y Java 476
Símbolos ANSI/ISO 5807-1985 y Raptor 477
Comparativo de comandos VB.NET, C#, C++ y Java 478
Generalidades de VB.NET, C#, C++ y Java 482

Índice temático 483

El curso *Aprenda Practicando Introducción a la Programación y su implementación usando VB.Net, C#, C++ y Java (1008)* es la alternativa para cubrir l a totalid ad de los temas contenidos en el presente libro, en un ambiente de aprendizaje i ntensivo, realizando todas las prácticas en el mínimo de tiempo.

Índice de prácticas

01.01 Estimando el tiempo de descarga de una película 9
01.02 Identificando los niveles de utilidad de los datos..... 9
01.03 Identificando un entorno operativo 10
03.01 Reconociendo la calidad de premisas y conclusiones 41
03.02 Analizando un silogismo y clasificando las premisas 48
04.01 Análisis de los nombres de variables 61
04.02 Identificando los tipos de datos idóneos para los datos..... 68
04.03 Análisis de economía de datos..... 69
04.04 Representación abstracta de dominios 70
05.01 Aplicación de operadores y sus reglas de precedencia 83
05.02 Representación de expresiones complejas operadores y elementos de lógica simbólica 85
06.01 Comprobando habilidades en el análisis de casos y el diseño de algoritmos.. 120
07.01 Comprobando habilidades en el análisis de casos y el diseño de algoritmos orientados a datos 165
08.01 Comprobando habilidades en el diseño de diagramas de flujo 198
09.01 Ejecución de comandos desde master console 216
09.02 Integrando instrucciones a un programa 222
09.03 Declaración de variables y arreglos, y uso de la instrucción assignment 227
09.04 Manejo de bucles usando Loop..... 230
09.05 Formas de ejecución de un programa en Raptor y el uso de pruebas de escritorio automáticas..... 233
09.06 Integrando peticiones de datos 235
09.07 Integrando condicionales y mostrado de datos..... 239
09.08 Manejo de puntos de interrupción (breakpoints) y comentarios..... 243
09.09 Creación y consumo de subcharts 246
09.10 Desarrollando un programa de regla de tres simple usando Raptor 259
09.11 Desarrollando un programa para el cálculo del área de un polígono irregular usando Raptor 259
09.12 Desarrollando un programa con bucles, condicionales y arreglos usando Raptor..... 261
10.01 Codificación de C1Mensaje en Visual Basic.NET 282
10.02 Codificación de C2Niveles en Visual Basic.NET..... 284
10.03 Codificación de C3Tabla en Visual Basic.NET..... 286
10.04 Codificación de C4MultiTabla en Visual Basic.NET 288
10.05 Codificación de C5Promedios en Visual Basic.NET..... 291
10.06 Desarrollando un programa de regla de tres simple usando Visual Basic.NET 294
10.07 Desarrollando un programa para el cálculo del área de un polígono irregular usando Visual Basic.NET..... 294
10.08 Desarrollando un programa con bucles, condicionales y arreglos usando Visual Basic.NET 296
11.01 Codificación de C1Mensaje en C# 315
11.02 Codificación de C2Niveles en C#..... 317
11.03 Codificación de C3Tabla en C#..... 319
11.04 Codificación de C4MultiTabla en C# 321
11.05 Codificación de C5Promedios en C# 324
11.06 Desarrollando un programa de regla de tres simple usando C#..... 327
11.07 Desarrollando un programa para el cálculo del área de un polígono

www.detodoprogramacion.com

irregular usando C# 328
11.08 Desarrollando un programa con bucles, condicionales y arreglos usando C# . 329
12.01 Codificación de C1Mensaje en C++ 349
12.02 Codificación de C2Niveles en C++ 350
12.03 Codificación de C3Tabla en C++ 352
12.04 Codificación de C4MultiTabla en C++ 355
12.05 Codificación de C5Promedios en C++ 357
12.06 Desarrollando un programa de regla de tres simple usando C++ 360
12.07 Desarrollando un programa para el cálculo del área de un polígono irregular usando C++ 361
12.08 Desarrollando un programa con bucles, condicionales y arreglos usando C++362
13.01 Codificación de C1Mensaje en Java 382
13.02 Codificación de C2Niveles en Java 383
13.03 Codificación de C3Tabla en Java 385
13.04 Codificación de C4MultiTabla en Java 388
13.05 Codificación de C5Promedios en Java 390
13.06 Desarrollando un programa de regla de tres simple usando Java 394
13.07 Desarrollando un programa para el cálculo del área de un polígono irregular usando Java 394
13.08 Desarrollando un programa con bucles, condicionales y arreglos usando Java396
14.01 Creando una clase y generando una instancia 413
14.02 Definiendo y utilizando propiedades 414
14.03 Creando una jerarquía de clases por herencia 422
14.04 Comprobando el polimorfismo basado en herencia con el uso de MyBase 424

Total de ejercicios: **62**

Prólogo

Iluso es pensar obtener resultados distintos,
haciendo las mismas cosas
Albert Einstein

Lo importante en un lenguaje es lo que se tiene que decir con él

En términos computacionales, supongo que pertenezco a lo que sería la tierra media. Una época en la cual si querías ver un cuadro dibujado en la pantalla, había que programarlo; una buena máquina tenía 20 MB en disco duro y un monitor que soportara 16 colores era un lujo que muy pocos privilegiados podían tener.

En esos días, como programador debías tener lógica de programación o te morías de hambre, dado que sin lógica era imposible desarrollar nada. Ahora las cosas han cambiado. Pasó el tiempo y las máquinas mejoraron, mejoraron las herramientas y los lenguajes también; ahora colocar un cuadro es tan fácil como arrastrar y colocar una figura en la pantalla, y el desarrollador no sabe qué sucede atrás, tan bambalinas. ¿Tengo algo en contra de la mejora continua de equipos, lenguajes y herramientas? Para nada. Vivo de ello.

Mi preocupación va en otro sentido. Pasan los años y como consultor ya me sé a la perfección los síntomas de una empresa con problemas de desarrollo de aplicaciones. Año con año sigo escuchando las mismas quejas: desfases en tiempo, desfases en presupuesto, y requerimientos mal entendidos pero bien desarrollados.

El común denominador es siempre un cliente descontento, antes como ahora.

Si las máquinas, los lenguajes y las herramientas de desarrollo han evolucionado tan radicalmente, y sin embargo seguimos teniendo los mismos problemas, entonces hemos estado buscando la solución a los problemas de desarrollo en el lugar equivocado.

Para identificar la solución a un problema basta con encontrar una situación libre de problemas, digámoslo así, un caso de éxito. Después analizamos nuestro caso de fracaso, y fácilmente detectamos las diferencias con el caso exitoso; corregimos el rumbo en nuestros casos de fracaso y corregimos la situación. Es una simple aplicación de mejores prácticas.

www.detodoprogramacion.com

Hemos identificado que los desarrollos condenados al fracaso tienen el común denominador de ser desarrollos poco estructurados y estandarizados, caóticos, en donde el rol principal lo juegan las máquinas y herramientas, y no la mente humana. El problema es que los requerimientos no los hace una computadora, sino una persona a la que llamamos cliente.

No conozco un cliente que se queje por haber obtenido exactamente lo que pidió al precio justo. Para que ello suceda hay que analizar sistemáticamente lo que quiere, definir sus requerimientos y validarlos con él antes que nada; cuando ya ha aprobado los requerimientos, es necesario reducir el caso real a pasos que se puedan automatizar, construir un modelo abstracto, identificar los datos del modelo abstracto, diagramar el modelo y darle coherencia y secuencia, miniespecificar lo que han de ser las actividades que el programa debe desarrollar, codificar el modelo usando un lenguaje de programación, compilar los programas y depurarlos, probar que la aplicación hace lo que debe, y finalmente presentarlo al cliente para su aceptación. Esto funciona, pero no es fácil de hacer por personas que no han tenido práctica en ello.

Este libro enseña al lector a realizar todas las actividades anteriores, aumentando la capacidad de éxito del desarrollador. Sólo será necesario aplicar muchas habilidades mentales que nada tienen que ver con la computación, como lo son la capacidad de análisis, de abstracción y de conclusión.

No será una tarea fácil para algunos. En estos tiempos las nuevas generaciones ya han sido evangelizadas por la televisión, en donde todo es inmediato. Cualquier problema que toma más de cinco minutos de análisis mental ya es un fastidio, y esa circunstancia ciertamente no ayuda.

Las herramientas de computación cada día simplifican los procesos de producción de programas, pero no nos engañemos: la solución de los problemas está en la forma en que éstos se plantean, y es en la fase de análisis en donde se realiza dicha actividad. Finalmente la solución óptima de los programas está en nuestro cerebro y en su asombrosa capacidad de analizar, abstraer y concluir de forma sistemática. Las herramientas de desarrollo sólo permiten implementar más rápido lo que ya habita en nuestra imaginación.

En el 2003 escribí un artículo por invitación del Colegio Interamericano, en Guatemala. Decidí llamar al artículo “Notas particulares para el programador frustrado”, dado que trataba de razonar las causas de la falta de efectividad en el aprendizaje de la programación. En ese artículo afirmaba que la lógica de programación no podía enseñarse, dado que existen muchas formas de razonar. Sigo en lo mismo: la lógica de programación no puede enseñarse, sólo puede aprenderse. Sólo es cuestión de pensar lógicamente en términos generales, y será cuestión de tiempo en que nuestra forma de razonar toque todos los aspectos de nuestra vida, incluso la programación.

Espero que este libro sea una fiesta del conocimiento, y no un funeral de vocaciones. La programación no es un arte, es una técnica. Cualquiera que pueda pensar podrá analizar, cualquiera que pueda analizar podrá generar algoritmos, quien pueda generar algoritmos podrá generar modelos de datos, y quien pueda generar modelos de datos podrá miniespecificar, codificar y probar programas. ¿Usted puede

www.detodoprogramacion.com

pensar? Si la respuesta es que sí, entonces no hay excusa. Sólo es necesario ponerse a trabajar.

Cuando Dios nos manda inspiración
es mejor que nos encuentre trabajando.
Pablo Picasso

Agradecimientos

Este libro es entrañable para mí, por muchas razones. Le debo tanto a la programación que 400 páginas son un homenaje injusto.

Con quien no debo ser injusto es con todos aquellos que forman un entorno en donde la escritura es posible.

A mi esposa Adriana, que me conoce a fondo y sabe sacar de mí la mejor persona que puedo ser, en todos los sentidos; agradezco además su ingenio para mantener a sana distancia a la pequeña Sara mientras escribo, lo cual es casi un deporte extremo.

A Isela Martínez, cuya ayuda incansable en la revisión editorial permite que escritos relativamente decentes puedan aspirar por un instante a ser escritos maravillosos y geniales.

A mis maestros, programadores de la vieja guardia: José Luis Ramos, Hugo Segundo y José Méndez; a mis amigos en la Universidad, Paco Salazar, Mary Araiza, Magda Madrigal, Eliud Palacios y Juan Enrique Saldaña; a Jorge Hernández y Elsa de la Fuente, del CSI en la Facultad de Ciencias Físico Matemáticas. A mis compañeros en el Gobierno Federal: Verónica López, Rey Manuel Martínez, y Aldo Avilés; a mis pupilos en la programación: Paco Guevara, Heriberto González, Maggy Córdoba, Antonio Ramírez, Carlos Eduardo Treviño, y muy especialmente a Víctor Manuel Ortega, que dio una revisada con ojos de alumno universitario al texto, evitando que el contenido tomara tintes filosóficos crípticos y poco entendibles (que vaya sí se me dan).

A los proveedores de música para programar: Diana Krall, The Gathering, Slayer, Iron Maiden, Tool, Mastodon. Qué tedioso programar sin música.

Gracias a Dios sobre todo, por dejarme ser feliz y no morir en el intento.

*Monterrey, Nuevo León, Junio **2006**.*

www.detodoprogramacion.com

¿Cómo utilizar este libro?

Este libro es una herramienta para la formación del pensamiento lógico y el razonamiento, aplicable a la función de programación de computadoras.

El libro está compuesto por material contenido en el curso de certificación 100 de Aprenda Practicando: Introducción a la programación y su implementación en Visual Basic.NET, C , Java y C .

El contenido está pensado para que el lector, en un período de 4 a 6 meses, aprenda las técnicas de análisis y abstracción de casos reales, su representación de forma técnica y su codificación.

Si se es autodidacta y se dedica tiempo completo a la realización de los ejercicios del libro, se puede aprender el contenido del libro en 2 meses, con el apoyo de las herramientas de Aprenda Practicando en Línea.

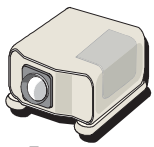
El soft are que se requiere está disponible en línea, en las siguientes rutas:

.NET Framework Redistributable Package 2.0 .NET Framework SDK 2.0	msdn.microsoft.com/
Borland C++ Compiler 5.5	www.borland.com/downloads/index.html
J2SE 5.0 JDK	www.sun.com/java/

www.detodoprogramacion.com

¿Cómo se componen los capítulos?

Los capítulos del presente libro tienen los siguientes elementos que fomentan el aprendizaje:



lpxx.ppt

- **Presentación profesional en Power Point.** Si se utiliza el presente libro para la exposición de cátedra o de cursos, la buena noticia es que no tendrá que desarrollar material de apoyo didáctico. En lugar de eso, podrá descargar de manera gratuita desde el sitio Web de Aprenda Practicando las presentaciones profesionales en Power Point que hemos desarrollado. Si las imprime como página de notas, adicionalmente obtendrá importante información para que la transmisión del conocimiento se lleve a cabo de manera efectiva. Cada capítulo señala, bajo el número de capítulo, el archivo que se deberá descargar para apoyar la exposición. Para descargar las presentaciones sólo es necesario ir al sitio http://www.aprendapracticando.com

- **Objetivo general y objetivos particulares.** Cada capítulo tendrá en su primera página un objetivo general, que representa la nueva habilidad que habremos adquirido al concluir el estudio del capítulo; además señalará los objetivos particulares, que representan en secuencia lógica los focos de atención para alcanzar el objetivo general.

- **Índice por capítulo.** Cada capítulo contiene su propio índice. Ubicar la información dentro de un capítulo resulta más fácil; además, el índice diferencia visualmente los temas de los ejercicios, a fin de que se pueda localizar a estos últimos de manera más fácil.

- **Notas informativas.** Cada capítulo contendrá notas adicionales, que profundizan en conocimiento general que no está disponible generalmente en la documentación del producto; además, se anotan aquí referencias históricas de los productos, así como consejos y advertencia que son para tomarse muy en cuenta, reduciéndose la curva de aprendizaje.

- **Videos explicativos.** Cada capítulo puede contener videos explicativos que abordan temas que es más fácil tratar de manera visual y dinámica, que textual y estática. Los videos están disponibles si se cursa el material en el entorno virtual de aprendizaje de Aprenda Practicando.

- **Gráficos ilustrativos.** Cada capítulo contendrá gráficos que permitirán ilustrar los temas tratados; a veces una imagen dice más que mil palabras.

- **Explicaciones textuales.** Cada capítulo contiene explicaciones profundas, que tratan el por qué de las cosas; esta es la principal diferencia con respecto a los manuales técnicos, que sólo explican cómo hacer las cosas, aunque el lector no entienda lo que está haciendo.

- **Ejercicios detallados, con explicaciones línea por línea y ejecución guiada.** La mayoría de los capítulos poseen ejercicios que permiten comprobar la teoría estudiada. Los ejercicios son desarrollados paso a paso; los programas pueden ser escritos aún sin disponer de herramientas visuales de desarrollo, y son ilustrados con líneas de código.

- **Terminología.** El texto introduce muchos conceptos nuevos que es necesario tener en mente. Cada capítulo tiene referido su propio índice de conceptos, por lo cual será muy fácil consultar el significado de los conceptos relevantes que es necesario saber para entender los temas.

- **Preguntas.** Cada capítulo incluye una serie de preguntas que tienen por objeto que el lector analice su propia opinión y experiencia respecto a lo que está aprendiendo. Las preguntas hacen referencias a los temas explicados en el capítulo, pero no cuestionan definiciones ni tópicos cuya respuesta está en el texto. Con las preguntas la idea es que piense respecto a lo que estudia.

- **Mapas mentales.** Los mapas mentales muestran en una sola imagen todos los temas ancla que se han cubierto en el capítulo. Actúan como resumen del capítulo, mismo que puede ser explorado de un vistazo.

- **Examen rápido.** Es la evaluación rápida que se debe resolver después de haber estudiado los temas, y de haber resuelto los ejercicios. Si se entendió cuáles eran los objetivos de estudio, y se contestaron las preguntas, se entendió la terminología, se realizaron los ejercicios y las prácticas individuales, es seguro que la evaluación siempre será contestada de una forma perfecta, con conocimiento de causa respecto a todo.

Sabemos de la enorme diversidad de personas. Cada persona tendrá su forma particular de aprender; algunos aprenden leyendo un resumen, otros viendo un video, otros leyendo, otros contestando preguntas, otros practicando, otros siendo cuestionados. Tratamos de brindar la mayor gama de herramientas, a fin de que cada quien aprenda como quiera, siempre y cuando aprenda.

En nuestro sitio eb encontrará además toda una plataforma de aprendizaje, denominado Aprenda Practicando en Línea que permitirá organizar el estudio y llevar un seguimiento del avance en el aprendizaje.

Estrategias de uso

Para el autodidacta...

La persona autodidacta es la que toma el aprendizaje en sus manos; no depende de las explicaciones de nadie, y por lo tanto tiene control total de su propio proceso de aprendizaje. A ese tipo de personas le sugerimos lo siguiente:

1. Todos los capítulos del 1 al tienen secuencia, por lo que recomendamos su lectura uno tras otro.
2. Atienda de manera secuencial cada capítulo del tema.
    a. Lea con detenimiento los objetivos del capítulo, para que tenga presente que eso es lo que hay que saber al concluir el capítulo.
    b. Si interrumpe la lectura del capítulo, cada vez que reinicie la lectura, lea de nuevo los objetivos. Es muy importante que los tenga presentes.
    c. Realice los ejercicios en el momento en que los encuentra en el texto.

d. Responda el examen rápido; si no contesta correctamente la mayoría de las preguntas, vuelva a revisar el capítulo, hasta que quede satisfecho.

3. Recomendamos, si así lo desea, que pruebe los cuestionarios y juegos que en línea se encuentran en www.aprendapracticando.com, para comprobar el grado de experiencia que ha adquirido.

Para el uso con grupos...

Si el libro se utiliza para la capacitación de grupos, sugerimos que se tomen en cuenta las siguientes consideraciones para alcanzar los objetivos (suponemos que usted es el instructor facilitador):

Al iniciar n te a

1. El instructor facilitador deberá dominar el tema a instruir; previo a la instrucción, deberá responder a las preguntas, realizar los ejercicios, realizar las prácticas individuales, y responder a la perfección el examen rápido.
2. Los equipos en los que se pretendan demostrar los ejercicios deberán estar preparados para el desarrollo las tecnologías que cubre el libro.
3. Las presentaciones profesionales en Po er Point están disponibles en el sitio eb de Aprenda Practicando (http://www.aprendapracticando.com); descargue las versiones más actualizadas, antes de cada curso.
4. Antes de revisar una parte del libro, explique la importancia del aprendizaje del tema. Genere el interés por el tema; sin interés no hay aprendizaje.
5. Si puede citar su experiencia personal en el uso del conocimiento del tema, es lo ideal para demostrar que la tecnología estudiada es útil. En la medida en que la audiencia piensa que el tema es importante, dedicará una mayor atención.

Para re isar ca a ca t lo

6. Solicite, antes de la sesión de instrucción, que los miembros del grupo lean el capítulo y resuelvan las preguntas al final del capítulo. Puede pedirlo como tarea escrita (de preferencia a mano, para contrarrestar el Copy Paste), para garantizar que los participantes leyeron con anterioridad.
7. Al iniciar cada sesión deberá recordar los objetivos del capítulo que se estudiará.
8. En caso de que se inicie capítulo, después de ver los objetivos, solicite a los miembros del grupo que lean las respuestas a las preguntas (punto 6); de preferencia ceda la palabra de tal forma que todas las personas participen.
9. Proyecte la presentación profesional que acompaña a cada capítulo; para un mejor aprovechamiento, recomendamos que para su uso personal imprima la

www.detodoprogramacion.com

presentación como página de notas, y lea las notas de cada una de las filminas, que le apoyarán con consejos y anotaciones relevantes.

10. Solicite a los alumnos que vean los videos que están disponibles en línea, y discuta su contenido.

11. En forma de exposición, solicite a los miembros del grupo que realicen los ejercicios hasta el final, cuando ya se haya expuesto toda la presentación.

12. Los miembros del grupo deberán realizar los ejercicios que el instructor facilitador les pida. Se deberá especificar con claridad cuál es el tiempo máximo de realización; es importante que los tiempos marcados se respeten.

13. El instructor facilitador, en caso de detectar dificultades en la realización de los ejercicios, deberá apoyar a los miembros del grupo, pero sin hacer el trabajo por ellos.

14. En caso de detectar un problema que sucede comúnmente en alguna práctica, es importante comentar con el grupo el problema y la solución.

15. Al concluir la realización de los ejercicios, el instructor facilitador comentará las experiencias más relevantes que haya notado; repasará por última vez los objetivos del capítulo y comentará cómo fue que se cubrieron.

16. Al finalizar los ejercicios, el instructor facilitador pedirá al grupo que contesten el examen rápido, de manera individual. Se darán 3 minutos para contestar; después, en voz alta planteará cada una de las preguntas, y revisará qué contestó el grupo. Después de oír las diferentes posturas, dirá cuál es la respuesta correcta y por qué. Los exámenes son una herramienta de aprendizaje más, no una base para el premio y el castigo.

17. El instructor facilitador podrá encargar las prácticas individuales como trabajo de refuerzo del conocimiento, ya sea para su desarrollo en sesión, o para que las personas desarrollen las prácticas en casa.

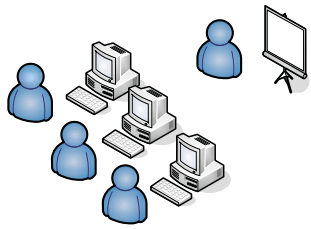
18. De ser posible, motive a los participantes a comprobar sus conocimientos en los diferentes cuestionarios y juegos que en línea se encuentran en www.aprendapracticando.com.

19. Si la infraestructura lo permite, puede solicitar a los miembros del grupo que vayan cubriendo la estrategia de aprendizaje de Aprenda Practicando en Línea.

Archivos complementarios del libro

En el sitio eb de Aprenda Practicando encontrará todos los archivos requeridos para el estudio de los temas del libro; además, podrá ver en línea los videos referidos en el texto. Si así lo desea, también podrá utilizar todas las herramientas de Aprenda Practicando En Línea (APE), que ayudarán al aprendizaje estructurado del curso.

www.detodoprogramacion.com



Todo el contenido de este libro, incluyendo los programas de implementación de los casos prácticos en Visual Basic.NET, C#, C++ y Java, pueden cubrirse en una semana, asistiendo al curso profesional:

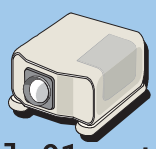
1008: Introducción a la Programación y su implementación usando Visual Basic.NET, C#, Java y C++.

Este curso es requerido para la certificación CAP-DNET (Certificación Aprenda Practicando – Desarrollador en Microsoft .NET), y puede cursarse en centros autorizados Aprenda Practicando.

ota i ortante Dependiendo de la demanda de los recursos educativos ofrecidos gratuitamente por Aprenda Practicando, y la respuesta de los lectores, estos pueden variar en su funcionamiento, agregarse o eliminarse, sin que sea necesaria notificación previa por escrito.

1

La computadora y los niveles de datos



lp01.ppt

Objetivos: Identificará la composición básica de una computadora y los niveles de datos que se pueden manejar utilizándola.

1. Aprenderá los componentes generales de un sistema de cómputo.
2. Aprenderá cuáles son los dispositivos que se pueden controlar mediante la programación.
3. Aprenderá cuáles son los tipos de programa que se pueden desarrollar.
4. Aprenderá cuáles son los niveles de datos.

www.detodoprogramacion.com

Contenido

Computadora 3
Tipos de dispositivos 4
Programas 5
Niveles de datos 6
En términos físicos 6
En términos de relevancia 8
Estimando el tiempo de descarga de una película 9
Identificando los niveles de utilidad de los datos 9
Identificando un entorno operativo 10
Manipulación de datos 11
Mapa mental del capítulo 13
Terminología 14
Preguntas 14
Examen rápido 15

La computadora, como todas las herramientas, no piensa.

Imagine alguien que quiere ser cirujano y no se ha tomado la molestia de aprender bien cómo se compone el cuerpo humano. No obstante ello, se compra el herramental médico (bisturí, sierra, hilo y agujas) y se dispone a meterle cuchillo al primer paciente que se cruce en su camino. Queda claro que a medida que abra a unos cuantos pacientes irá aprendiendo cómo es el cuerpo humano, pero es seguro que no nos gustaría ser nosotros uno de sus pacientes.

Lo mismo sucede con muchas disciplinas, aunque sin los efectos drásticos del ejemplo anterior. La gente quiere lo que llamo yo "pensar con las manos", es decir, empezar a actuar sin saber qué está haciendo.

Precisamente uno de los principales problemas al aprender a programar es que se olvida el fin último de la programación, que es darle órdenes a una herramienta llamada computadora. Mucha gente no sabe qué es una computadora ni cómo se compone, y aún así se aventura a intentar programar; como es de esperarse el resultado es bastante mediocre.

Si además tomamos en cuenta que hoy en día las computadoras son muy variadas, más que nunca debemos hacernos a la idea que estamos tratando con muchas herramientas que debemos conocer de forma particular. Las computadoras se parecen en esencia, pero no todas son iguales. Médicamente hablando, los programadores de computadoras se parecerían más a un veterinario que a un médico de humanos, en el sentido que tendrán que aprender la composición de diferentes sujetos de trabajo, para luego poder manejarlos eficientemente.

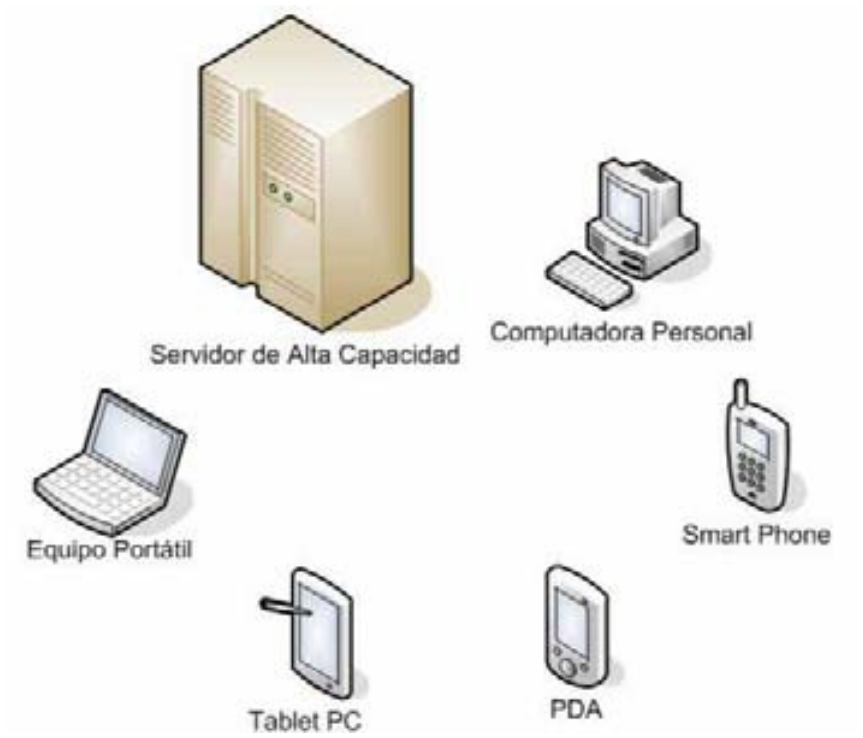
Computadora

Iniciamos definiendo a un *sistema* como el conjunto de elementos interrelacionados entre sí para alcanzar un mismo fin.

Podemos definir a una *computadora* como un sistema de dispositivos electrónicos, mecánicos y magnéticos que sirven para procesar y almacenar datos. Las computadoras reciben nombre comercial dependiendo de su tamaño o su uso, aunque en esencia son lo mismo. Tenemos las computadoras de escritorio, las computadoras portátiles, los PDA y celulares, etcétera.

Figura 01.01

Diferentes computadoras.



Una computadora es un sistema, dado que los dispositivos que la componen trabajan de forma interrelacionada, con el objetivo de apoyar el fin último de la computadora: procesar y almacenar datos.

Tipos de dispositivos

En su concepto básico, una computadora se forma de dispositivos de naturaleza electrónica, mecánica o magnética, y se clasifican de la siguiente manera.

- *Dispositi os de entrada.*

  Son los dispositivos que permiten proporcionar a la computadora los datos a procesar o almacenar, o bien, indicarle a la computadora la ejecución de acciones. Ejemplos típicos de este tipo de dispositivos son el teclado y el ratón.

- *Dispositi os de salida.*

  Son los dispositivos que permiten comunicar resultados de procesamiento al usuario de la computadora. El típico dispositivo de salida es la pantalla del monitor.

- *Dispositi os de procesamiento.*

  Son los dispositivos que se encargan del procesamiento de los datos. El principal dispositivo de procesamiento es lo que se conoce como *nidad Central de rocesamiento* (*C Central rocessing nit*), que contiene dos cosas muy importantes: la memoria RAM y el microprocesador.

  La *memoria* (*andom ccess emor*) es de vital importancia, pues es el área de almacenamiento temporal en el que los programas deben ser cargados previamente a fin de que el microprocesador pueda trabajar con ellos. Nada puede trabajarse en la computadora si no está antes en la RAM. La memoria RAM se divide en *posiciones de memoria* a las que se puede tener acceso a través de *direcciones de memoria* (*emor ddress*), en las cuales se puede alojar (*allocate*) información en forma de datos de diferente tipo.

  Con respecto al *microprocesador*, es el que se encarga del trabajo de cómputo del equipo, a través de dos unidades internas: unidad aritmética y unidad lógica (de ahí que se le llame *unidad aritmético lógica*).

  La unidad aritmética se encarga de las operaciones de suma, resta, división y multiplicación, mientras que la unidad lógica se encarga de resolver comparaciones y decisiones, conduciendo la información por los canales adecuados.

- *Dispositi os de almacenamiento permanente.*

  Son los dispositivos en los cuales podemos almacenar datos de manera persistente, es decir, que no se pierdan al momento de apagar el equipo.

- *Dispositi os peri éricos.*

  Son los dispositivos de salida que no forman parte de la computadora, pero que pueden conectarse a ésta para ampliar su funcionalidad. Ejemplo de este tipo de dispositivo son las impresoras y los equipos biométricos.

Conocer los diferentes tipos de dispositivo es de vital importancia, ya que aprender a programar es aprender a controlar los dispositivos mediante instrucciones.

Al conjunto de dispositivos físicos que componen una computadora se le da el nombre de *hard are*.

Programas

El hard are por sí solo no sirve para nada. Es necesaria la existencia de programas que le indiquen a los dispositivos qué hacer.

Podemos identificar los siguientes tipos de programas controlando a una computadora.

- *istema operati o.*

  Es el conjunto de programas y datos que permiten reconocer, identificar y utilizar los dispositivos de una computadora. Generalmente, los otros tipos de programas se comunican con el sistema operativo, y es éste el que comunica las órdenes a los dispositivos.

- *rogramas aplicati os.*

  Es el conjunto de programas y datos que utilizan de forma genérica las capacidades de la computadora, para realizar tareas específicas. Generalmente se trata de programas comerciales como Microsft ord, Macromedia Flash, etcétera; se caracterizan por permitir la creación de archivos autónomos de trabajo.

- *istemas de in ormación.*

  Es el conjunto de programas y datos que permiten utilizar las capacidades de procesamiento y almacenamiento de la computadora, con el fin de generar, manipular y divulgar información. Generalmente se trata de programas desarrollados en la misma organización, o adquiridos a terceros. Se caracterizan por no producir archivos autónomos de trabajo, sino por consumir bases de datos.

A todos los elementos físicamente intangibles que participan en el funcionamiento de la computadora se les da el nombre de *so t are*.

A un determinado hard are funcionando a través de un determinado soft are, se le da el nombre de *entorno operati o* o *plata orma operati a*. Los entornos operativos son muy útiles al momento de verificar compatibilidad de los programas y los equipos; generalmente cuando usted requiera soporte le preguntarán qué equipo (marca y modelo) está usando, y qué sistema operativo (marca, versión, actualizaciones) es el que tiene instalado. En otras palabras, para dar soporte se requiere conocer el entorno operativo. También cuando compre soft are encontrará seguramente los requerimientos mínimos del programa, que generalmente harán referencia a una plataforma operativa específica.

Niveles de datos

Los datos manejados por una computadora pueden ser entendidos de dos maneras distintas: desde el punto de vista físico y desde el punto de vista de relevancia.

El punto de vista físico tiene que ver con el consumo de áreas de almacenamiento que los datos ocupan, mientras que desde el punto de vista de relevancia, lo que importa es el significado y el valor que el usuario le otorga a los datos.

En términos físicos

Internamente las computadoras manejan los datos en formato binario (0 y 1).

Las computadoras desconocen si lo almacenado es importante o no, e incluso desconocen el significado de lo que almacenan.

Bits. Se conoce como *bit* a los ceros y unos manejados internamente por la computadora. Los bits son la unidad básica de almacenamiento y comunicación de información dentro de la computación, y son procesados con suma eficiencia por el procesador de la computadora. Una operación tan simple como una suma o una comparación pueden consumir capacidad de procesamiento, de ahí que los procesadores actuales deben ser capaces de realizar millones de operaciones por segundo.

Se tienen los siguientes símbolos y *unidades de medida para los bits*.

ilobit	bit	2^{10} bits
megabit	Mbit	2^{20} bits
gigabit	Gbit	2^{30} bits
terabit	Tbit	2^{40} bits
petabit	Pbit	2^{50} bits

Bytes. Un conjunto de 8 bits hacen un *b te*, que en esencia equivalen a un símbolo o caracter legible a los humanos.

¿Por qué son 8 bits los que componen un byte, y no 7 ó 9?

La respuesta es una cuestión de economía. En los orígenes de la computación, dado lo limitado de los recursos de las computadoras, había que decidir por el mínimo número de bits que compondrían a un símbolo, pero que al mismo tiempo permitiera un conjunto de combinaciones suficientes para manejar todos los símbolos utilizados en un idioma.

Dado el manejo binario de las computadoras, las opciones eran las siguientes:

2^1 2
2^2 4
2^3 8
2^4 16
2^5 32
2^6 64
2^7 128
2^8 256
2^9 512
2^{10} 1024
...

Se llegó al consenso que 2^8 proporcionaba un juego de caracteres de 256 símbolos, en los cuales era posible almacenar las letras mayúsculas (A, B, C, etc.), minúsculas (a, b, c, etc.), y símbolos especiales (, , , , , , etc.) Esto al menos en su versión occidental.

A cada juego de 256 símbolos se le denomina *p gina de códigos*. Las páginas de códigos de un solo byte contienen un máximo de 256 valores. Una página de códigos con un límite de 256 caracteres no puede adaptarse a todos los idiomas porque para ello sería necesario utilizar mucho más de 256 caracteres. Por lo tanto, los distintos alfabetos utilizan páginas de códigos separadas. Hay una página de códigos para el griego, otra para el japonés, otra para el español, etcétera.

Aunque un código tan reducido puede ser suficiente para idiomas como el Ha aiano, que sólo tiene 12 caracteres, puede ser muy limitado para otros idiomas más complejos. Las páginas de códigos de un solo byte resultan insuficientes para la mayoría de los idiomas asiáticos, ya que éstos utilizan normalmente más de 5,000 caracteres basados en el Chino. Afortunadamente ya se han desarrollado páginas de códigos de doble byte, con el fin de obtener compatibilidad con estos idiomas.

NOTA

En la actualidad, dado que las aplicaciones pueden ser multilingües gracias a la globalización, 256 caracteres no son suficientes para representar todos los diferentes símbolos que se pueden utilizar. Afortunadamente el almacenamiento ya no es un problema tan grave, por lo que propuestas como *UNICODE* permiten asignar una codificación basada en 2 bytes, con lo que es posible almacenar 2^{16} combinaciones, lo que es bastante. Investiga más acerca de UNICODE.

Los bytes, tienen los siguientes símbolos y unidades de medida.

ilobyte	B	2^{10} bytes
megabyte	MB	2^{20} bytes
gigabyte	GB	2^{30} bytes
terabyte	TB	2^{40} bytes
petabyte	PB	2^{50} bytes

En términos de relevancia

s m s importante la imaginación que el conocimiento

Albert Einstein

Por *alor* entendemos la utilidad o deleite que nos proporciona algo, al menos en nuestra percepción.

Somos las personas quienes le otorgamos a los datos el carácter de valiosos o irrelevantes. Los datos pueden tener diferente nivel de utilidad para las personas, y esta utilidad generalmente está asociada al significado que tienen para nosotros.

Podemos proponer los siguientes niveles de los datos, en cuanto a la relevancia que tienen para nosotros:

- *Dato.*

  Es el conjunto de representaciones simbólicas no significativas, dado que no tenemos la capacidad de reconocerles un significado más allá del que tienen los símbolos que componen el dato.

- *In ormación.*

  Es el conjunto de datos que en determinada cantidad y forma aumenta el conocimiento o reduce la incertidumbre respecto a un sujeto, evento o circunstancia. La información sólo podrá ser considerada información si tenemos capacidad de reconocerla.

- *Conocimiento.*

  Es la información, cuando tenemos capacidad para usarla en actividades del plano real y limitado. El conocimiento tiene naturaleza práctica.

- *Imaginación.*

  Es la información, cuando tenemos capacidad para usarla en actividades del plano irreal e ilimitado. La imaginación tiene naturaleza teórica.

Ejercicio 01.01

Estimando el tiempo de descarga de una película

Resuelva el siguiente problema.

1. Usted tiene un módem de 56 bit segundo, que utiliza para conectarse a Internet por línea telefónica.
2. Como la línea telefónica no es muy buena, el módem sólo aprovecha el 40 de su velocidad de transmisión.
3. Usted encuentra en un sitio en Internet la película *tar ars pisodio III*, disponible para descargarse.
4. Si la película mide 4GB ¿cuántas horas tomará descargar el archivo?

horas.

FIN DEL EJERCICIO ✱

Ejercicio 01.02

Identificando los niveles de utilidad de los datos

Identifique si se trata de datos, información, conocimiento e imaginación.

1. Un niño pequeño está aprendiendo a leer. Se encuentra una revista especializada en neurocirugía, y la lee en voz alta.

2. Usted se memoriza un libro de química para un examen. Si bien no entiende lo que está leyendo, le servirá para acreditar el examen.

3. Albert Einstein trabaja en su mente con modelos que ni siquiera puede comprobar en la vida real; no puede desplazar un cuerpo a la velocidad de la luz, por ejemplo, pero matemáticamente

www.detodoprogramacion.com

comprueba su teoría de la relatividad y revoluciona la física moderna.

4. Usted estudió un libro de nutrición, para verificar la combinación de proteínas, carbohidratos y grasas que debería comer, de acuerdo a su peso, género y edad. Gracias a ese estudio ha podido nivelar su peso y estar más saludable.

FIN DEL EJERCICIO ✱

Ejercicio 01.03

Identificando un entorno operativo

Identifique el entorno operativo del equipo que utilizará para trabajar en este curso. El entorno operativo es la combinación de hard are y soft are.

1. Respecto al Hard are, especifique la capacidad de su equipo:

Procesador (modelo)	
Velocidad del procesador (MHz)	
Memoria RAM (MB)	
Capacidad de disco (GB)	
Cuenta con unidad CD (si no)	
Cuenta con unidad DVD (si no)	
Cuenta con tarjeta de video (si no)	
Cuenta con tarjeta de red (si no)	

2. Respecto al soft are, especifique la capacidad de su equipo:

Sistema operativo	
Versión	
Edición	
ltimo Service Pac instalado	

FIN DEL EJERCICIO ✱

Manipulación de datos

De los niveles de relevancia de los datos, sólo nos interesa la información, ya que los programas que realicemos generalmente procesarán datos con el fin de almacenarlos o de producir información.

Las acciones que es posible realizar con los datos son los siguientes:

- *ecopilación de datos.*

  Cuando los datos ya existen, sólo se captan para su procesamiento o almacenamiento. La recopilación puede hacerse preguntándolos al usuario o tomándolos de archivos. Ejemplo típico de la recopilación puede ser una pantalla en Internet que nos solicita nuestro correo electrónico.

- *rocesamiento de datos.*

  Cuando los datos se someten al uso de operadores aritméticos lógicos, con el fin de crear datos nuevos o modificar los existentes. Un ejemplo típico de esto es cuando corregimos nuestra información personal (por ejemplo nuestro nic name) en un programa de mensajería electrónica.

- *eneración de datos.*

  Cuando se generan datos nuevos y desconocidos a partir de datos conocidos y existentes (datos conocidos procesamiento datos nuevos). Un ejemplo típico de esta acción es cuando se genera un monto total, como resultado de la suma de los precios de varios artículos.

- *Di ulgación de datos.*

  Cuando los datos existen y se hacen llegar a los usuarios que deben tenerlos. En esta acción no se genera nada nuevo, pues sólo se trata de exposición de datos. Un ejemplo típico de esto son las páginas eb que muestran información a través de Internet.

En las aplicaciones complejas conviven todas las acciones en un momento determinado. Tome en cuenta por ejemplo una página eb en donde se venden canciones MP3. De inicio se realiza divulgación, dado que muestra la información

www.detodoprogramacion.com

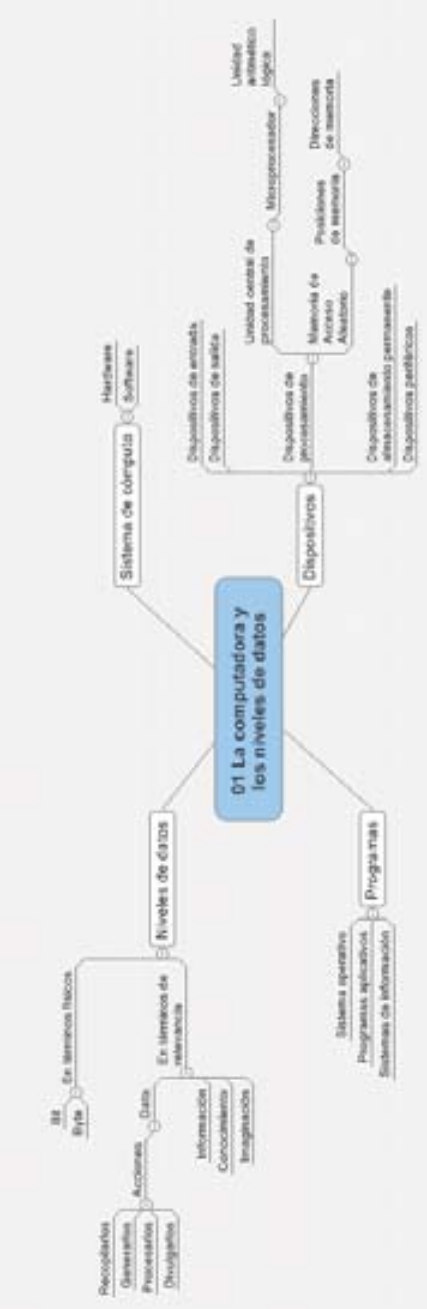
de los nuevos discos que están a la venta. Si a usted le interesa alguna de las canciones, la selecciona para comprarla. La página realiza recopilación de datos al mantener internamente la información de las canciones que ha seleccionado, y además preguntándole la información de su tarjeta de crédito. La página también procesa y genera nuevos datos a partir del precio de las canciones que quiere comprar, calculándole los impuestos y cargos de envío. Finalmente se procesa información, haciendo el cargo a su tarjeta de crédito.

Es muy probable que al someter a los datos a cualquiera de las acciones anteriores, éstos asuman el carácter de información. Recordemos que dicho carácter tiene que ver con el valor que tienen los datos para nosotros, y no con las acciones que hayamos aplicado sobre ellos.

Las acciones que realizamos con los datos generalmente se llevarán a cabo con la asistencia de equipo de cómputo, a través de la cual se automatizan de forma rápida y confiable.

La *in orm tica* es la ciencia que estudia la forma automatizada en que se recopila, procesa, genera y divulga la información.

Mapa mental del capítulo



Terminología

Bit, 6
Byte, 6
Central Processing Unit, 4
Computadora, 3
Conocimiento, 8
CPU, 4
Dato, 8
Direcciones de memoria, 4
Dispositivos de almacenamiento permanente, 4
Dispositivos de entrada, 4
Dispositivos de procesamiento, 4
Dispositivos de salida, 4
Dispositivos periféricos, 4
Divulgación de datos, 11
Entorno operativo, 5
Generación de datos, 11
Hardware, 5
Imaginación, 8
Información, 8
Informática, 12
Memoria RAM, 4
Memory Address, 4
Microprocesador, 4
Página de códigos, 7
Plataforma operativa, 5
Posiciones de memoria, 4
Procesamiento de datos, 11
Programas aplicativos, 5
Random Access Memory, 4
Recopilación de datos, 11
Sistema, 3
Sistema operativo, 5
Sistemas de información, 5
Software, 5
UNICODE, 7
Unidad aritmético lógica, 4
Unidad central de procesamiento, 4
Unidades de medida para los bits, 6
Valor, 8

Preguntas

1.1 Usted se encuentra un libro de energía termo nuclear, que en la portada dice "Contiene la información de los estudios más recientes respecto al tema". ¿Considerando sus conocimientos respecto al tema, considera que el libro contiene información o no?

1.2 Usted trabaja en un restaurante que tiene terminales de punto de venta con pantallas equipadas con sistema touch screen ¿dichos dispositivos son de entrada o de salida?

1.3 ¿Por qué la memoria RAM no forma parte de los dispositivos de almacenamiento permanente?

1.4 ¿Por qué cree que Albert Einstein consideraba más importante la imaginación que el conocimiento?

1.5 Mencione un ejemplo de cada una de las acciones posibles de realizar con los datos, distintas a las mencionadas en el libro.

Examen rápido

Parte 1 : Marque co n una cruz el inciso que considere que responde a la pregunta.

1) La memoria RAM forma parte de los
(a) Dispositivos de almacenamiento permanente
(b) Dispositivos de procesamiento
(c) Dispositivos periféricos

2) Son los programas que se caracterizan por generar archivos autónomos de trabajo.
(a) Sistemas operativos
(b) Programas aplicativos
(c) Sistemas de información

3) Son el conjunto de dispositivos físicos que componen una computadora.
(a) Hard are
(b) Soft are
(c) Dispositivos

4) Nivel de datos de utilidad práctica en el plano real y limitado.
(a) Dato
(b) Información
(c) Conocimiento

5) Acción asociada a los datos que utiliza mayormente la unidad aritmético lógica.
(a) Generación de datos
(b) Procesamiento de datos
(c) Recopilación de datos

Parte 2: Coloque en la línea la letra «F» si la sentencia es falsa, y «V» si la sentencia es verdadera.

6) Un bit se compone de 8 bytes.

7) La informática trata la automatización de las acciones realizadas con los datos.

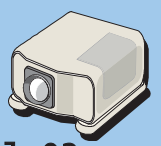
8) Para efectos prácticos, la imaginación es más importante que el conocimiento.

9) Los dispositivos periféricos son una parte importante de las computadoras.

10) Con UNICODE es posible almacenar más de 256 caracteres por página de código.

2

Lenguajes de programación y su clasificación



lp02.ppt

Objetivos: Aprenderá qué son los lenguajes de programación, su clasificación, y la forma en que permiten producir código máquina.

1. Aprenderá qué son los lenguajes de programación, e identificará sus elementos.
2. Aprenderá cuáles son los tipos de código que pueden generarse utilizando los lenguajes de programación.
3. Aprenderá el proceso de convertir código fuente a código máquina.
4. Aprenderá las diferentes clasificaciones de los lenguajes de programación.

www.detodoprogramacion.com

Contenido

Lenguajes de programación 20
Tipos de código 23
Construyendo secuencias binarias ejecutables 24
Compiladores 25
Utilidad del código intermedio 25
Clasificaciones de los lenguajes de programación 26
Clasificación de los lenguajes en cuanto a su generación 26
Clasificación de los lenguajes en cuanto a su nivel 28
Clasificación de los lenguajes en cuanto a su propósito 28
Clasificación de los lenguajes en cuanto a su orientación 29
Mapa mental del capítulo 31
Terminología 32
Preguntas 32
Examen rápido 33

Lo importante no es el lenguaje que uno habla

sino lo que uno tiene que decir cómo lo dice.

Si la computadora pudiera leer nuestra mente y realizar de forma automática las acciones que queremos que haga el mundo sería otro (empezando porque no existirían programadores, ni este libro).

Afortunadamente eso sólo ocurre en las películas de ciencia ficción. En la vida real las cosas suceden de una forma bastante más primitiva: la computadora sólo hace aquello que el programador le indica que haga. Si el programador es bueno, la computadora es útil; si el programador es malo, la computadora es un gasto inútil.

Lo verdaderamente importante es que el programador sepa darle instrucciones a la computadora, pidiéndole lo que quiere que haga. Aquí surge un nuevo problema: la computadora sólo entiende ceros y unos, mientras que nosotros hablamos nuestro idioma nativo. El reto es encontrar una forma de expresión que tanto la computadora como nosotros entendamos.

Todos ya sabemos que las computadoras trabajan con un sistema numérico basado en ceros y unos, denominado *sistema binario*. Es así por que a través de ceros y unos se pueden representar estados de encendido y apagado, que tal como la clave Morse, puede utilizarse para transmitir mensajes, lo que en computación llamaríamos *paquetes de datos*.

Internamente las computadoras sólo entienden de *secuencias de n meros binarios*, que no son otra cosa que paquetes de datos reconocidos por el sistema operativo como instrucciones que provocan comportamientos específicos en los dispositivos.

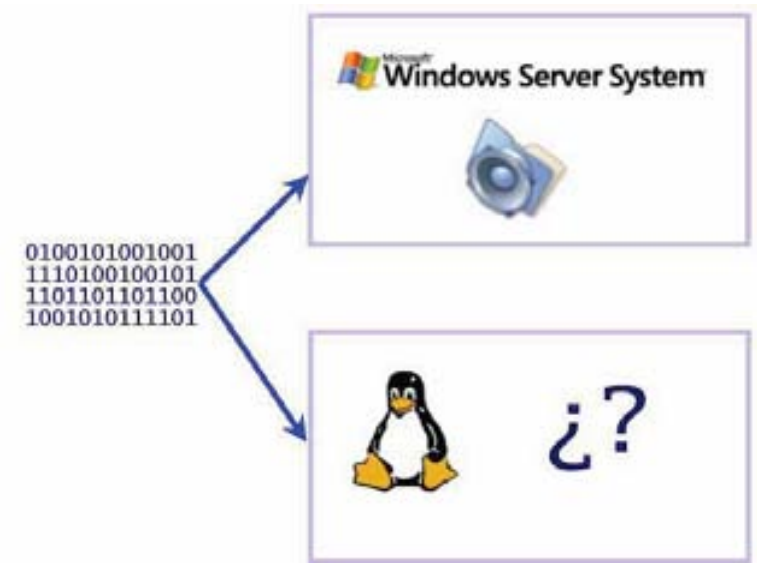
Una determinada secuencia de ceros y unos representará una acción específica para una determinada plataforma operativa, pero la misma secuencia de ceros y unos puede no significar nada para otra plataforma operativa. A este fenómeno se le da el nombre de *incompatibilidad*, que se define como la incapacidad de equipos o dispositivos de reconocer un conjunto de instrucciones o paquete de datos. Los programas que funcionan en una plataforma, por ejemplo indo s 2003, seguramente no funcionarán en otra, como puede ser UNI o Linux. En ese sentido, los equipos serán incompatibles. Es importante aclarar que no siempre las diferencias entre plataforma operativa producen incompatibilidad; un equipo corriendo en indo s 2003 y otro ejecutando indo s P, siendo plataformas operativas distintas, pueden leer prácticamente los mismos paquetes de datos, contenidos en archivos que ambos sistemas pueden reconocer.

Dependiendo la necesidad que se tenga, deberá optarse por la plataforma operativa más conveniente. Seguramente si su negocio es el diseño gráfico seleccionará un equipo Mac, dado que esa plataforma operativa tiene grandes ventajas en ese tipo de trabajo; si lo que desea es realizar trabajo de oficina, y tiene la necesidad de compartir mucha información con mucha gente, seguramente seleccionará plataforma indo s, por ser la que tiene más difusión en el mercado (*mar et share*).

No sea de aquellos que se desgarran las vestiduras por tal o cual plataforma, del tipo "amo a Linux odio a indo s"; sinceramente hay cosas más interesantes en qué apasionarse. Cada plataforma tiene sus ventajas y desventajas, y debemos tener la cabeza lo suficientemente fría para elegir lo más conveniente para una situación dada, más allá de nuestras preferencias personales. Aquél que es obsesivo con una plataforma específica lo único que hace es auto limitarse en sus posibilidades profesionales.

Figura 02.01

Una secuencia binaria que para Windows es un sonido, para Linux puede no significar nada.



Al conjunto de instrucciones representadas como secuencias de números binarios que una determinada plataforma de equipo reconoce, provocando en él una determinada acción, se le llama *lenguaje m quina*. Nuestro trabajo como programadores es generar código máquina.

Lenguajes de programación

Los *lenguajes de programación* son estructuras simbólicas que nos permiten disponer de los dispositivos de una computadora.

Los lenguajes de programación tienen lo siguientes elementos.

- *structura del código*.

  Es la forma general en que deberá escribirse un programa, con el fin de delimitar los bloques de código que los componen, en una forma ordenada y sistemática. Por *bloque de código* debemos entender al conjunto de líneas de código que pueden ser consideradas como unidad independiente de ejecución. La estructura del código evita que se escriban porciones de código en lugares inadecuados, en los que no tiene sentido que se escriba código.

- *alabras reser adas*.

  Son palabras con significado dentro del lenguaje de programación, que no podrán ser utilizadas más que en el sentido original en el que fueron creadas. En realidad se trata de símbolos que provocarán un comportamiento predecible. Generalmente son palabras en inglés, que en el caso de ser verbos están en infinitivo; palabras como PRINT, o DELETE son comunes en todos los lenguajes.

- *presiones*.

  Son valores expresados de forma entendible para el lenguaje de programación. Generalmente los datos de tipo numérico no tienen problemas, ya que se escriben como normalmente los entendemos. Las expresiones textuales, llamadas *cadenas*, son secuencias de caracteres generalmente delimitadas por comillas simples o dobles, dependiendo del lenguaje. Datos como las fechas ya son más complejos y se estudiarán cuando se trate el tema de las expresiones.

- *mbolos especiales*.

  Son los caracteres (generalmente no más de dos), que provocan un comportamiento predecible dentro de un programa. A esta categoría pertenecen típicamente los operadores aritméticos (, , ,) y de comparación (, , ,).

- *inta is.*

  Según la Real Academia Española, la sintaxis es la parte de la gramática que enseña a coordinar y unir palabras para formar oraciones y expresar conceptos. En el caso de los lenguajes de programación, la sintaxis es la forma ordenada en que se deben especificar las palabras reservadas, las expresiones y los símbolos especiales, a fin de que tengan significado para el lenguaje de programación. Generalmente el no respetar las reglas de sintaxis de un lenguaje provoca errores en los programas, o simplemente no producen los comportamientos que se esperaban.

- *em ntica.*

  Son las reglas que determinan el significado que ha de dársele a una palabra reservada, expresión o símbolo especial, dependiendo del contexto de sintaxis en que se encuentran. Un ejemplo en el que la semántica toma relevancia es cuando se utiliza en algunos lenguajes el símbolo de igualdad ( ); dependiendo de la sintaxis utilizada el símbolo de igualdad puede asignar un valor, o bien indicar una comparación en la que deseamos saber si un valor es igual a otro.

El objetivo primordial de un lenguaje de programación es hacernos la vida más sencilla, proporcionándonos formas humanamente comprensibles de construir secuencias de números binarios reconocidos por un entorno operativo, a través de símbolos y palabras equivalentes.

Sin la existencia de lenguajes de programación avanzados, sería imposible la elaboración de programas tan útiles y complejos como Microsoft ord o Microsoft Excel. Los millones y millones de secuencias binarias que tendríamos que proporcionar a la computadora para obtener la funcionalidad que nos dan tales programas excederían nuestro tiempo de vida.

NOTA

Los lenguajes de programación tienen un asombroso parecido con el lenguaje que utilizamos para hablar, dado que tienen los mismos elementos. Un ejemplo: en algunas comunidades en donde se utiliza como lengua de expresión el inglés y el español de forma indistinta, por ejemplo en Miami, Florida, se tiene la tendencia a modificar la sintaxis; podemos escuchar que preguntan "¿Qué tu piensas?", cuando la estructura correcta sería "¿Qué piensas tú?". No emitamos la crítica fácil sin analizar el punto. Tomando en cuenta que la misma pregunta en inglés sería "What do you think?", entendemos que están homologando una misma sintaxis para dos lenguajes distintos, lo que facilita su aprendizaje. Los lenguajes de programación están haciendo lo mismo, de tal forma que ya no es extraño ver programadores dominando más de un lenguaje.

En este libro se tratan los lenguajes de programación Visual Basic.NET, C , C y Java. Cada uno de ellos tiene su propia identidad y sus legiones de seguidores.

Este sencillo programa se encarga de solicitarnos un dato textual, para luego mostrarlo.

Visual Basic.NET

```
1   Imports System
2
3   Module C1Mensaje
4      Sub Main()
5         Dim Mensaje As String
6         Console.Write("Escribe un mensaje a mostrar: ")
7         Mensaje = Console.ReadLine()
8         Console.WriteLine(Mensaje)
9      End Sub
10  End Module
```

C#

```
1   using System;
2
3   class C1Mensaje
4   {
5           static void Main()
6           {
7                   String Mensaje;
8                   Console.Write("Escribe un mensaje a mostrar: ");
9                   Mensaje= Console.ReadLine();
10                  Console.WriteLine(Mensaje);
11          }
12  }
```

C++

```
1   #include <iostream>
2   using namespace std;
3
4   void main()
5   {
6      string Mensaje;
7              cout << "Escribe un mensaje a mostrar: ";
8      cin >> Mensaje;
9      cout <<  Mensaje << endl;
10  }
```

Java

```
1   import java.lang.*;
2   import java.io.*;
3
4   class C1Mensaje
5   {
6           public static void main(String args[])
7                   throws java.io.IOException
8     {
9                   BufferedReader entrada = new
10                          BufferedReader(new InputStreamReader(System.in));
11                  String Mensaje="";
12                  System.out.println("Escribe un mensaje a mostrar: ");
13                  Mensaje=entrada.readLine();
14                  System.out.println(Mensaje);
15          }
16  }
```

Dependiendo su gusto personal puede seleccionar el lenguaje que más le acomode. Lo importante no es el lenguaje que utiliza para decir las cosas, sino lo que dice.

Ninguno de los programas que ilustramos significa nada para la computadora, dado que no son secuencias de números binarios. Aún falta entender el proceso de compilación, que se encarga de traducir nuestros programas a un formato entendible para la computadora, es decir, en secuencias binarias ejecutables.

Tipos de código

Codificar es representar un mensaje de forma sistemática, atendiendo leyes o normas específicas. En el caso de la programación, lo que se quiere representar son acciones y actividades que queremos que la computadora realice. Lo hacemos de forma sistemática porque agrupamos palabras reservadas, estructuras básicas, símbolos especiales, que unidos nos permiten representar la realidad que queremos automatizar.

Utilizamos los lenguajes de programación para representar de forma codificada las actividades que deseamos automatizar. Lo que nosotros escribimos es incomprensible para la computadora, ya que es necesario que se traduzca en secuencias de números binarios que pueda entender.

Los tipos de código que existen son los siguientes.

- *Código uente.*

  Es el código tal y como nosotros lo escribimos. Los ejemplos que hemos visto en este capítulo son ejemplos de código fuente. Nosotros como programadores podemos leerlos e interpretarlos sin la necesidad de una

computadora. El código fuente puede ser ambiguo dado que muchas personas pueden escribir programas diferentes que hagan lo mismo; en otras palabras, representan de diferente forma lo mismo. En código fuente, dos programas que hacen lo mismo pero que están en diferente lenguaje de programación son enteramente distintos.

- *Código intermedio.*

  Es el código que resulta de haber sometido al código fuente a la fase de análisis del proceso de compilación. Este proceso lo libera de ambig edades, de tal forma que se eliminan los errores y excesos del programador, dejando una representación abstracta del programa. El código intermedio ya no puede ser leído o interpretado por el programador, pero aún no puede ser interpretado por la computadora. Este código es independiente de la máquina. El código intermedio puede ser compilado o interpretado; es compilado cuando se somete a la fase de síntesis del proceso de compilación, generando código máquina a partir de él; es *código interpretado* cuando es leído por un programa intérprete que provoca el comportamiento del programa, sin que nunca exista realmente una versión en código máquina del código fuente. En ocasiones al código intermedio interpretado se le da el nombre de *código objeto*.

- *Código m quina.*

  Es el código que resulta de haber sometido al código intermedio a la fase de síntesis del proceso de compilación. En este proceso, el código intermedio da lugar al código máquina, que son secuencias de números binarios que la computadora interpreta. Cabe aclarar que el código máquina no puede ser interpretado por el programador, y en la mayoría de los casos tampoco puede ser interpretado por computadoras que funcionan en diferente ambiente operativo que la computadora que se utilizó para su creación. Este código es dependiente de la máquina. Al programa que contiene el código máquina y la información suficiente para que el sistema operativo reconozca que se trata de código máquina, se le llama *programa ejecutable*. En el ambiente indo s DOS, la extensión de estos archivos es E E.

Construyendo secuencias binarias ejecutables

Los lenguajes sirven para generar programas que serán ejecutados en la computadora con el fin de que realicen tareas específicas.

Los programas, tal y como los escribimos, con instrucciones, comandos y estructuras que podemos leer, reciben el nombre de código fuente. El código fuente no significa nada para la computadora, pues se trata de un conjunto de palabras y símbolos; como sabemos la computadora sólo entiende secuencias de números binarios que reconoce como órdenes.

Compiladores

Los *compiladores* son programas que leen un programa escrito en un lenguaje, al que llamamos código fuente, y lo traduce a un nivel de abstracción entendible por la computadora.

Los compiladores le traducen los programas fuentes a la computadora, utilizando para ello un proceso en dos fases: análisis y síntesis.

Fase de análisis. El proceso de *an lisis* consiste en descomponer un todo en sus partes a fin de entender mejor el todo. En términos de compilación se llama *ase de an lisis* a la parte del proceso que divide el programa fuente en sus elementos (instrucciones, estructuras de control, declaraciones, operadores, etcétera) validando que esté sintáctica y semánticamente correcto, generando una representación abstracta del código, al que llamamos código intermedio.

Este código intermedio es una abstracción del código fuente, que dice de qué forma se debe comportar el programa. Debido a que el código intermedio aún no es código nativo, es independiente del equipo. Generalmente el compilador que realiza únicamente la fase de análisis es conocido como *anali ador de código*, o *parser*.

Fase de síntesis. Por otro lado, el proceso de *s ntesis*, es lo opuesto al análisis, y se encarga de generar el código máquina. El código objeto que se genera de la compilación no siempre es ejecutable; puede ser que el código objeto esté formado por secuencias de números binarios que el sistema reconoce como órdenes; en este sentido, el código objeto será código nativo, y en consecuencia será ejecutable pero dependiente del equipo que lo reconoce.

Por otro lado, existe la posibilidad de que el código objeto sea una forma depurada de código intermedio; de ser así, se requerirá someter a dicho código objeto a otro proceso de compilación cuyo producto sí sea código nativo.

Se tienen dos tipos de compiladores: compiladores en una fase y compiladores en dos fases. Los *compiladores de dos ases* producen código máquina a partir del código fuente; el compilador realiza tanto la fase de análisis como la de síntesis, por lo cual este tipo de compiladores no generan código intermedio. Los *compiladores de una sola ase* se encargan de la realización de una sola fase del proceso de compilación (análisis o síntesis); el compilador que realiza la fase de análisis producirá código intermedio, mientras que el compilador que realiza la fase de síntesis producirá código máquina.

Utilidad del código intermedio

Si bien es cierto que un compilador puede generar código nativo a partir del código fuente, también es cierto que existen algunas ventajas de que el proceso de síntesis no genere código nativo:

- El código intermedio es independiente del equipo; compilar hasta código intermedio permite disponer de un nivel abstracto del código fuente, independiente del equipo.

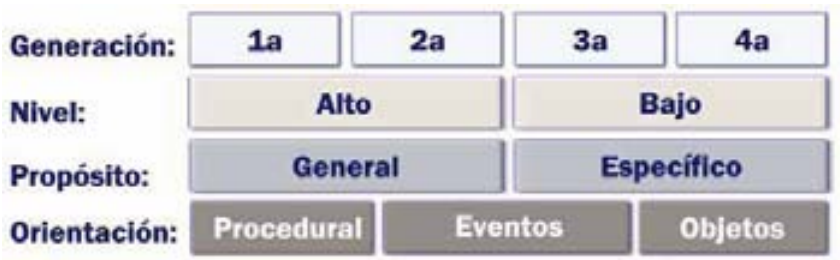
- Como el código intermedio no es código nativo, éste no es definitivo. Es posible aplicar sobre el código intermedio optimizadores de código, que tomen ventaja de los recursos existentes en el equipo al momento de la ejecución, y no al momento de la compilación.

Clasificaciones de los lenguajes de programación

Los lenguajes de programación pueden clasificarse de diferentes maneras.

Figura 02.02

Clasificaciones de los lenguajes de programación.



Clasificación de los lenguajes en cuanto a su generación

La *generación* a la que pertenecen los lenguajes indica la capacidad de representación que tienen los símbolos que lo componen. Entre más secuencias de números binarios represente cada símbolo del lenguaje, mayor será su generación.

Lenguajes de primera generación. En los inicios de la computación, las instrucciones se le proporcionaban a la computadora en lenguaje máquina, en formato de secuencia binaria, en obvia proporción de 1 a 1 (una secuencia binaria, una instrucción a la computadora).

Para desarrollar en los lenguajes de primera generación era necesario conocer qué representaba para la computadora cada una de las secuencias binarias que utilizaríamos; prácticamente, sólo el constructor de la computadora podía programarla. Esto planteaba una enorme limitación, ya que la capacidad humana para estar preparando las instrucciones, nos llevaba a la elaboración de programas bastante rudimentarios, por ejemplo una suma o una resta. Elaborar un programa compuesto por millones de secuencias de números binarios era imposible.

En resumen, en la primera generación se proporcionaban secuencias binarias directas, que pasaban a ejecución en la computadora sin interpretación o manipulación alguna.

Lenguajes de segunda generación. La mayor limitación de la primera generación era que las secuencias binarias eran difíciles de entender. En la segunda generación se recurrió al uso de *términos mnemotécnicos* (propios de un sistema para mejorar la memoria; del griego *mnémo*, *memoria*); la proporción de

instrucciones y símbolos seguía siendo de 1 a 1, pero con la ventaja de que ya no era necesario aprender secuencias binarias, sino palabras, generalmente abreviadas.

En resumen, en la segunda generación se proporcionaban mnemotécnicos que la herramienta de desarrollo traducía a secuencias binarias, que pasaban a ejecución en la computadora.

Lenguajes de tercera generación. Si en la segunda generación era posible representar una secuencia binaria a través de una palabra, también debería ser posible representar varias secuencias binarias a través de una palabra. Bajo esta premisa, nacen los lenguajes de tercera generación, como BASIC, COBOL, PASCAL, C, etcétera. Con estos lenguajes, la proporción de instrucciones y símbolos era de muchas a 1, lo que significó un avance exponencial en el uso de lenguajes. Además de la proporción instrucción símbolo, otro rasgo característico de esta generación es que la codificación, aunque más sencilla, seguía siendo realizada por el ser humano, es decir, por los programadores.

En resumen, en la tercera generación se programan instrucciones que la herramienta de desarrollo interpreta y traduce en una o más secuencias binarias, que pasan a ejecución en la computadora.

Lenguajes de cuarta generación. Algunos lenguajes y herramientas de desarrollo comenzaron a integrar sistemas de macroinstrucciones y generadores de código, automatizando el proceso de proporcionar a la computadora instrucciones de bajo nivel.

Las *macroinstrucciones* son instrucciones de nivel superior, que al ejecutarse producen los efectos equivalentes de un conjunto más amplio de instrucciones del mismo lenguaje, pero de nivel inferior.

Los *generadores de código*, por otro lado, son herramientas que con poca interacción del programador generan el código que desarrollaría ciertas tareas rutinarias, dejando al programador la codificación de aquellas reglas específicas del negocio, que no podían ser realizadas de manera automática.

Las macroinstrucciones y los generadores de código proporcionan el rasgo característico de los lenguajes de cuarta generación: la codificación es realizada por el ser humano, pero también por el lenguaje o por la herramienta misma. La codificación es un proceso asistido.

En resumen, en la cuarta generación se programan macroinstrucciones e instrucciones; la herramienta de desarrollo interpreta y traduce las macroinstrucciones en instrucciones de más bajo nivel, las junta con las instrucciones que el programador haya enviado, las complementa con instrucciones que genera de manera automática, para después interpretarlas y traducirlas en secuencias binarias, que pasan a ejecución en la computadora.

NOTA

Algunos sostienen la existencia de 5GL (Lenguajes de quinta generación); en nuestra opinión creemos que es un ardid mercadotécnico, al menos por ahora. Analizando la evolución de las generaciones de los lenguajes, sólo se consideraría como de quinta generación aquél sistema que eliminara la necesidad del ser humano de codificar, es decir, la existencia de un motor verdaderamente inteligente que aceptara sólo reglas de negocio, y que con base a ello, desarrollara todo el código. Hasta el momento no hemos visto una herramienta de desarrollo que así lo haga.

Clasificación de los lenguajes en cuanto a su nivel

El *ni el* de los lenguajes está asociado al número de plataformas en las que pueden ejecutarse los programas desarrollados en un determinado lenguaje. Como ya vimos, las computadoras sólo actúan en respuesta a secuencias binarias; cada plataforma de computadoras responde a las secuencias binarias que reconoce como instrucciones.

Lenguajes de alto nivel. Un lenguaje o herramienta de desarrollo es de alto nivel si existe la posibilidad de generar, a partir de un mismo código fuente, secuencias binarias que sean reconocidas por varias plataformas de computadoras. Lenguajes como C, COBOL, y PASCAL, pueden servir en plataformas distintas, como indo s, UNI y Linux, con un mayor o menor grado de portabilidad.

La *portabilidad* es la capacidad de los lenguajes y herramientas de desarrollo para generar productos que se ejecuten en diferentes plataformas, preferentemente sin cambios en la codificación.

Lenguajes de bajo nivel. Un lenguaje o herramienta de desarrollo es de bajo nivel si no existe la posibilidad de generar, a partir de un mismo código fuente, secuencias binarias que sean reconocidas por varias plataformas de computadoras.

Clasificación de los lenguajes en cuanto a su propósito

El *propósito* de los lenguajes está asociado a las ramas del conocimiento humano que las aplicaciones desarrolladas en dichos lenguajes pueden cubrir.

Lenguaje de propósito específico. Los lenguajes de propósito específico son aquellos que únicamente permiten desarrollar aplicaciones que cubren una determinada rama del conocimiento humano. Un ejemplo de este tipo es el lenguaje COBOL, que fue pensado para aplicaciones de negocios; desarrollar una compleja aplicación de graficación vectorial en dicho lenguaje puede resultar, si no es que imposible, sí algo muy inadecuado.

Lenguaje de propósito genera l. Los lenguajes de propósito general, por otra parte, permiten desarrollar aplicaciones que prácticamente cubren todas las ramas del conocimiento. Un ejemplo de esto es BASIC, que igual permite desarrollar una aplicación científica, como de negocios o de bases de datos.

Clasificación de los lenguajes en cuanto a su orientación

La *orientación* de los lenguajes tiene que ver con la forma en que se estructuran internamente las instrucciones en el código fuente y los programas.

Programación procedural. La programación procedural, llamada también de procedimientos, implica que las instrucciones deben ser ejecutadas de manera secuencial, una tras otra, tal y como fueron especificadas.

Este esquema permite saltos de control, es decir, transferir el control del programa de una línea a otra (GOTO); los programadores experimentados saben que hay ocasiones en que los saltos de control se realizan de manera tan caótica, que resulta difícil seguir la huella de las actividades a realizar, pues el flujo del programa no respeta una secuencia lógica.

La reutilización del código en este esquema sólo se da mediante la definición de procedimientos que pueden ser llamados a ejecución, o bien, repitiendo secuencias de instrucciones.

Este tipo de programación tiene la ventaja de ser bastante lógica (siempre y cuando no se abuse de los saltos de control), e incluso se recomienda para el aprendizaje de la programación.

Una desventaja de este tipo de programación es que la reutilización del código es rudimentaria y redundante, ya que si un mismo procedimiento aplica con pequeñas diferencias a dos circunstancias distintas, es necesaria la existencia de dos procedimientos diferentes: el comportamiento de los procedimientos es estricto y específico para una tarea. Otra desventaja es que los procedimientos secuenciales obligan a que la interfaz de las aplicaciones también sea secuencial.

Programación Orientada a Eventos (Event Driven Programming). La palabra *evento* proviene del latín *evenire*, que quiere decir suceder, producirse; en materia de programación, indica que la utilización de la interfaz de usuario provoca que suceda la ejecución de un procedimiento. Un ejemplo clásico de un evento es hacer clic en un botón de la interfaz; el usuario hace clic, luego entonces, pasa algo.

Como podrá darse cuenta, la programación orientada a eventos es en el fondo procedural, con la diferencia que la ejecución de los procedimientos no se realiza de manera secuencial, sino aleatoria, de acuerdo a la forma en que el usuario en contacto con la interfaz provoque los eventos.

Un *procedimiento de evento* es la secuencia de instrucciones que se ejecutarán cuando un usuario provoque un evento.

La ventaja de este tipo de programación es que el uso de la interfaz de las aplicaciones es menos estricto, ya que el usuario puede utilizarla sin orden alguno, y ello no perjudica la secuencia de ejecución del programa. La desventaja de este esquema es que sigue siendo en el fondo procedural, y por tanto tiene la misma

desventaja de dicho modelo, pues cada circunstancia distinta requiere su propio procedimiento.

Es importante señalar que un evento no sólo puede ser causado por el usuario en su interacción con la interfaz, sino que también puede ser causado por la aplicación, en interacción consigo misma.

Programación orientada a objetos (POO | Object Oriented Programming | OOP). Los *objetos* son entidades encapsuladas de código y datos, que a partir de datos de entrada que reciben, proporcionan una funcionalidad determinada.

En la orientación a objetos no importan las instrucciones y las secuencia de las mismas, ya que lo que verdaderamente importa es la funcionalidad de los objetos, sus entradas y sus salidas. Al hacer uso de un objeto, en realidad no sabemos si éste a su vez hizo uso de otros objetos, y éstos de otros que incluso se pueden encontrar en otro equipo de cómputo; como podrá ver, el orden secuencial de las instrucciones es totalmente irrelevante: no importa cómo o a través de qué se obtienen resultados. Lo importante es que se obtengan.

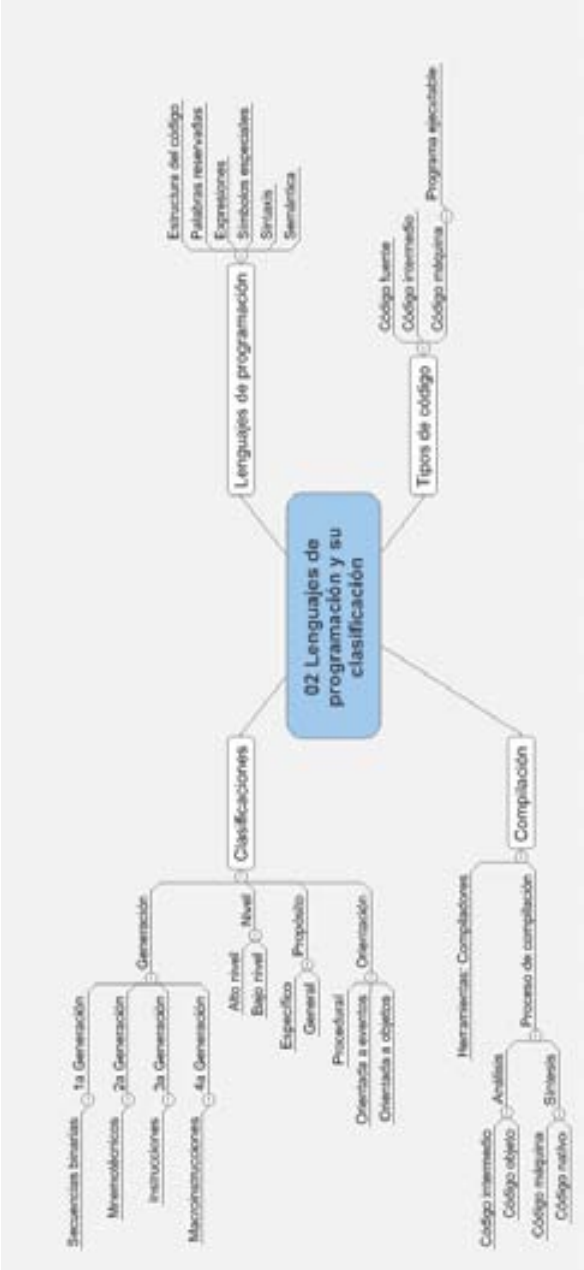
El tema de la programación orientada a objetos es tan interesante como extenso, y requiere un análisis particular; por el momento bastará saber que uno o más objetos pueden asociarse como respuesta a un evento determinado, eliminando la necesidad de los procedimientos de evento y obteniendo así todas las ventajas de dicho estilo de programación. Además, la programación orientada a objetos permite la reutilización de código de manera dinámica, ya que los objetos se adaptan a las circunstancias específicas de ejecución, permitiendo que un solo objeto maneje diferentes escenarios.

Una de las características más notables de la programación orientada a objetos es que permite la herencia entre objetos: un objeto particular puede heredar toda la funcionalidad de un objeto general, reutilizando el código de éste; con ello se ahorra muchas horas de trabajo destinado a reescribir código y a controlar modificaciones de código fuente.

Cuando los objetos ya están creados, es común agruparlos en bibliotecas o librerías de objetos.

Una de las desventajas del modelo de programación orientada a objetos es que resulta un poco más complejo de aprender; además, requiere mucho más planeación y técnica de programación para ser utilizado. Es muy bueno que la funcionalidad y el código puedan reutilizarse y heredarse, pero así como se pueden heredar cosas buenas, también se pueden heredar cosas mal hechas, por lo que debe ser más cauteloso.

Mapa mental del capítulo



Terminología

Análisis, 25
Analizador de código, 25
Bloque de código, 20
Cadenas, 20
Código fuente, 23
Código intermedio, 24
Código interpretado, 24
Código máquina, 24
Código objeto, 24
Compiladores de dos fases, 25
Compiladores de una sola fase, 25
Compiladores, 25
Estructura del código, 20
Evento, 29
Expresiones, 20
Fase de análisis, 25
Generación, 26
Generadores de código, 27
Incompatibilidad, 19
Lenguaje máquina, 20
Lenguajes de programación, 20
Macroinstrucciones, 27
Nivel, 28
Objetos, 30
Orientación, 29
Palabras reservadas, 20
Paquetes de datos, 18
Parser, 25
Portabilidad, 28
Procedimiento de evento, 29
Programa ejecutable, 24
Propósito, 28
Secuencias de números binarios, 18
Semántica, 21
Símbolos especiales, 20
Sintaxis, 21
Síntesis, 25
Sistema binario, 18
Términos mnemotécnicos, 26

Preguntas

2.1 En su opinión, ¿qué tan posible es que existan lenguajes de quinta generación?

2.2 En su opinión, sólo a simple vista, ¿qué lenguaje considera más sencillo? Explique por qué.

2.3 En su opinión, ¿qué es mejor, un lenguaje de alto nivel o uno de bajo nivel? Explique por qué.

2.4 No existe en el mundo un lenguaje de programación universalmente aceptado. Muchos son los programadores en .NET, muchos en Java y muchos en C . ¿Considera que la rivalidad entre los grupos de programadores de uno y otro lenguaje tiene que ver con las capacidades técnicas de los lenguajes, o cree que tiene que ver con aspectos más allá de lo técnico? Explique su respuesta.

2.5 Nosotros pensamos que la programación orientada a objetos es mejor que la programación procedural. ¿Está de acuerdo? Explique su respuesta.

Examen rápido

Parte 1 : Marque co n una cruz el inciso que considere que responde a la pregunta.

1) Elemento de un lenguaje de programación que nos dice la forma ordenada en que se deben especificar los demás elementos, podrán ser reconocidos por el lenguaje, y tener significado.
(a) Semántica
(b) Sintaxis
(c) Estructura

2) Son los dos tipos de código intermedio que existen.
(a) Código compilado y código interpretado
(b) Código objeto y código nativo
(c) Código ejecutable y código interpretado

3) Son los programas que llevan a cabo el proceso de análisis y síntesis de un programa.
(a) Intérpretes
(b) Programa ejecutable
(c) Compiladores

4) Es la clasificación de lenguajes que tiene que ver con el número de plataformas en las que puede ejecutarse un programa desarrollado en un determinado lenguaje.
(a) Generación
(b) Nivel
(c) Orientación

5) Es un rasgo característico de la POO.
(a) Modularidad
(b) Capacidad de compilación
(c) Herencia

Parte 2: Coloque en la línea la letra «F» si la sentencia es falsa, y «V» si la sentencia es verdadera.

_____ **6)** Las diferencias existentes entre plataformas operativas siempre provoca incompatibilidad.

_____ **7)** La fase de análisis en el proceso de compilación es la encargada de producir código máquina.

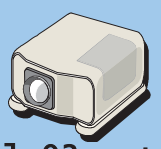
_____ **8)** El parser se encarga de la fase de síntesis del proceso de compilación.

_____ **9)** Los compiladores de dos fases generan código intermedio.

_____ **10)** El código fuente es el código más entendible para los humanos.

3

Introducción a la lógica de programación



lp03.ppt

Objetivos: **Aprenderá los elementos de la lógica y su aplicación en las labores de programación de computadoras y la automatización de procesos.**

1. **Aprenderá la teoría general de la lógica.**
2. **Aprenderá en qué consiste la lógica de programación.**
3. **Aprenderá a ubicar la finalidad de enunciados lógicos dentro de un proceso computarizado.**
4. **Aprenderá los principios fundamentales en la elaboración de enunciados y su relación con los datos y los procesos.**

www.detodoprogramacion.com

Contenido

Lógica 37
Silogismos, proposiciones y premisas 37
Calidad de las premisas y las conclusiones 39
Reconociendo la calidad de premisas y conclusiones 41
Lógica de programación 42
Aplicación de la lógica en la programación 43
Finalidad de los enunciados en el proceso 44
Principios en la elaboración de enunciados 46
Principios relacionados con los datos de entrada 46
Principios relacionados con el proceso 47
Principios relacionados con los datos de salida 47
Analizando un silogismo y clasificando las premisas 48
Mapa mental del capítulo 50
Terminología 51
Preguntas 51
Examen rápido 53

Entre la ignorancia y el saber sólo hay trabajo.

Cuando decimos que una persona es "inteligente" o "lista", ¿por qué lo decimos?

Lo más seguro es que no hayamos sometido a las personas a un examen para determinar su coeficiente intelectual. Simplemente, en el día a día, vemos que sobresalen gracias a la forma en que piensan y hacen las cosas, y les otorgamos la etiqueta de inteligentes. seamos sinceros: todos queremos ser considerados inteligentes.

Decimos que una persona es inteligente porque procesa las ideas más rápido que el promedio de las personas, y por consiguiente, puede actuar de forma más acertada y más rápida. Cuando los demás apenas están entendiendo un problema o situación, esa persona ya tiene una o más alternativas de solución. Podríamos decir que dentro de su mente esa persona cuenta con caminos mentales más cortos y seguros para llegar al mismo lado que los demás, y por eso siempre se encuentra en situación de ventaja permanente.

Podemos decir que la gente es inteligente dependiendo de los conocimientos que tienen. Eso no es del todo correcto: dos personas con iguales calificaciones académicas resolverán de diferente forma los problemas que se les presenten en la vida real.

Está comprobado que no siempre los alumnos con mejores calificaciones académicas son los más listos a la hora de enfrentar la vida. Incluso hay ocasiones en que una

persona con muchos conocimientos tarda más en resolver un problema, precisamente porque tiene tantos conocimientos que termina confundido entre tantas posibles soluciones.

Queda claro que el secreto no está sólo en la cantidad de conocimientos que tenemos, sino también en la forma en que utilizamos el conocimiento. En ese sentido, lo ideal es tener muchos conocimientos y saber qué hacer con ellos.

Imagine que puede aprender a utilizar mejor los conocimientos que tiene en la cabeza y que podrá pensar de forma más rápida y precisa. Mejor aún: imagine que las ideas que genere siempre serán correctas y bien entendidas por los demás.

Precisamente eso es lo que vamos a aprender en este libro, tomando como pretexto a la programación de computadoras.

Lógica

Para entender a la lógica es necesario definir dos actividades que ha venido realizando desde que nació: Razonar y Procesar.

Razonar es ordenar ideas para llegar a una conclusión.

Procesar es ejecutar una serie de actividades interrelacionadas entre sí que pretenden un objetivo común.

La *lógica* es la ciencia que estudia el *proceso* de *razonar*. En otras palabras, es la ciencia que estudia las actividades que realizamos para ordenar nuestras ideas y llegar a una conclusión.

La ventaja de la lógica es que estudia un proceso; si aprendemos bien el proceso podremos repetirlo muchas veces, repitiendo también los resultados obtenidos. Si aprendemos a razonar de forma eficiente habremos aprendido la forma de cómo aprovechar el poco o mucho conocimiento que tengamos. Los beneficios serán incalculables, ya que con pocas cantidades de conocimiento seremos capaces de generar muchas conclusiones y alternativas.

Ahora nuestro reto es educar al pensamiento para que razone de acuerdo a ciertas reglas; con ello podremos ser más eficientes al momento en que ordenamos nuestras ideas y sabremos cómo utilizarlas mejor. En otras palabras, seremos más lógicos.

Silogismos, proposiciones y premisas

Un *silogismo* es un argumento que consta de tres o más proposiciones, la última de las cuales se deduce de las otras. Por *argumento* entendemos un sistema de enunciados expresados en un lenguaje determinado, donde uno de los enunciados es una conclusión, y el resto son premisas.

Un *enunciado*, llamado también *proposición*, es una expresión ling ística que establece un pensamiento completo. Los enunciados pueden ser interrogativos, imperativos y declarativos; los que nos interesa estudiar son los *enunciados*

declarativos, que son aquellos sobre los que podemos afirmar su verdad o falsedad, es decir, un *valor de verdad*.

Una proposición termina siendo una afirmación o negación construida en forma de sentencia (sujeto y predicado). En sí es una propuesta.

Este es un ejemplo de un silogismo:

A)

Todos los perros son mamíferos.

Fido es perro.

Δ Fido es mamífero.

A las dos primeras proposiciones de un silogismo se les llama *premisas*, debido a que anteceden a la última proposición. A la primera proposición se le llama *premisa mayor*, mientras que a la segunda se le llama *premisa menor*. A la última proposición, misma que se infiere de las primeras, se le llama *conclusión*. El símbolo Delta (Δ) suele leerse como "por lo tanto", "entonces".

En nuestro ejemplo A, la premisa mayor nos proporciona información respecto a una circunstancia general (clase zoológica para todos los perros), la segunda premisa nos proporciona información respecto a una circunstancia particular (Fido y su condición de perro), y la conclusión es una inferencia obvia, ya que la característica señalada en términos generales (ser mamífero) aplica a todos los perros, y por tanto a Fido también. El ejemplo A tiene un orden tal que permite concluir casi de forma inequívoca una realidad.

Esa perfección no siempre se cumple.

¿Nunca le ha sucedido que platica con alguien, y a pesar estar hablando de algo sencillo no le entiende nada? Lo más probable es que no esté siendo lógico.

Vea las siguientes variaciones del silogismo:

B)

Fido es perro.

Todos los perros son mamíferos.

Δ Fido es mamífero.

C)

Todos los perros son mamíferos.

Fido es mamífero.

Δ Fido es perro.

En el caso del silogismo B la conclusión es obvia, aunque para algunos es más difícil llegar a la conclusión, en comparación con el silogismo A. El caso del silogismo C tiene serios problemas, dado que Fido puede ser mamífero, pero ello no necesariamente implica que sea perro.

Calidad de las premisas y las conclusiones

La calidad de las premisas depende de su apego a la realidad, y pueden ser del siguiente modo:

FALSAS (F)	VERDADERAS (V)
Si la premisa no representa a la realidad.	Si la premisa representa a la realidad.

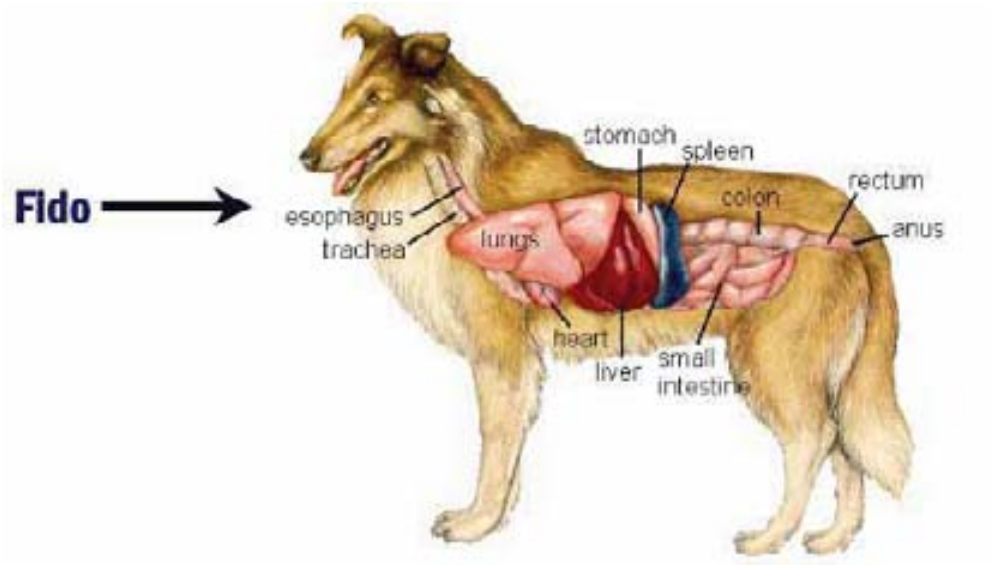
Estos modos también le aplican a la conclusión, en su calidad de proposición.

La calidad de una conclusión depende del grado en que es determinada por las premisas, y puede ser del siguiente modo:

Incorrectas (I)	Correctas (C)
Si las premisas no proporcionan todos los elementos para inferir la conclusión.	Si las premisas proporcionan todos los elementos para inferir la conclusión.

Si desea que todo el mundo entienda lo que dice y la forma en que piensa, preocúpese por utilizar en su pensamiento y en su expresión premisas verdaderas, y llegar siempre a conclusiones correctas. Un argumento es correcto si y sólo si no es posible que sus premisas sean verdaderas y su conclusión falsa.

Si analizamos dos de nuestros ejemplos, en este contexto, tenemos:



A)

Todos los perros son mamíferos.	(V)
Fido es perro.	(V)
Δ Fido es mamífero.	(V) y (C)

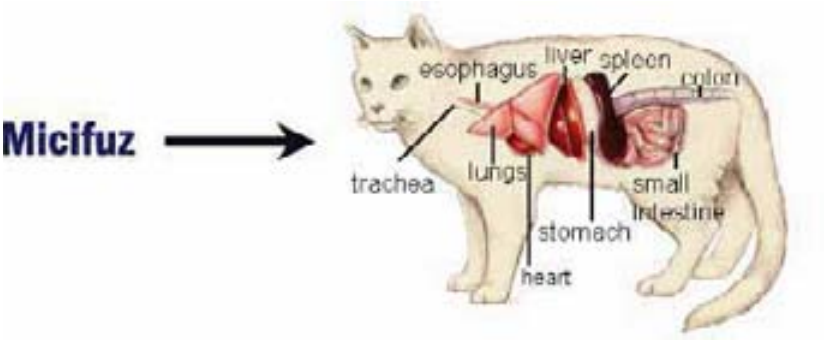
C)

Todos los perros son mamíferos.	(V)
Fido es mamífero.	(V)
Δ Fido es perro.	(V) e (I)

Ejercicio 03.01

Reconociendo la calidad de premisas y conclusiones

En este ejercicio se analizará una serie de silogismos, determinando si sus proposiciones son Verdaderas (V), Falsas (F), Correctas (C) o Incorrectas (I). El contexto es el siguiente:



1. Coloque en la línea de la izquierda la letra V, F, C, o I, dependiendo si la proposición es Verdadera (V), Falsa (F), Correcta (C) o Incorrecta (I).

   Todos los gatos son mamíferos. _____

   Micifuz es perro. _____

   Δ Micifuz maúlla. _____

2. Escriba un silogismo de tipo (V), (V), (V)(C).

3. Escriba un silogismo de tipo (V), (V), (VI).

4. Escriba un silogismo (F), (F), (V)(C).

FIN DEL EJERCICIO ✱

Lógica de programación

La *lógica de programación* es la habilidad de pensar de manera razonada, sistemática y ordenada, que nos hace capaces de inferir algoritmos abstractos y soluciones a problemas de negocios, que fácilmente puedan ser implementadas a través de código escrito en un lenguaje de programación determinado.

La lógica de programación considera dos procesos fundamentales de la computación: el procesamiento y el almacenamiento. El procesamiento tiene que ver con someter a ciertos datos de entrada a un proceso aritmético lógico que produzca ciertos datos de salida.

Por definición, un *dato de entrada* es aquél que se le proporciona a un proceso para que genere datos de salida. Un *proceso* es el conjunto de operaciones aritmético lógicas a las que son sometidos los datos de entrada para producir los datos de salida. Los *datos de salida* son la resultante de haber sometido datos de entrada a un proceso. Como datos de salida se consideran también las nuevas versiones de datos existentes que hayan sido generadas como producto del proceso; un ejemplo típico de esto es la modificación de un registro en una base de datos: el registro ya existe, pero un proceso puede modificarlo.

www.detodoprogramacion.com

Un *programa de cómputo* es la implementación, a través de un lenguaje de programación, de un sistema que recopila datos de entrada y los procesa con el fin de obtener datos de salida. Los datos de entrada pueden ser preguntados al usuario, pueden ser tomados de fuentes externas al programa, o ser verdades generalmente aceptadas (datos preexistentes).

Estas definiciones parecen obvias y triviales, pero no lo son; de hecho, son el elemento de la lógica de programación.

Aplicación de la lógica en la programación

Para entender a la lógica de programación es necesario que entendamos que con los datos es posible representar enunciados, que harían las veces de preposiciones.

Considere que el área de un triángulo rectángulo se calcula multiplicando la base por la altura, y dividiendo el resultado entre dos.

rea (Base Altura) 2

Traducido a un silogismo sería:

ilo is o A

El área es igual a base por altura entre dos	(V)
La base es 10 y la altura es 5	(V)
Δ El área es 25	(V) (C)

Qué pasaría si propusiéramos lo siguiente:

ilo is o

El área es igual a base por altura entre dos	(V)
La base es 10	(V)
Δ El área es 25	(V o F) (I)

En el caso del segundo silogismo no tenemos manera de saber si la conclusión es verdadera o falsa, dado que desconocemos el valor de la altura. Como no es posible llegar a la conclusión con los datos disponibles, la conclusión no puede ser más que incorrecta.

Vea cómo la primera premisa termina siendo el proceso, la segunda los datos de entrada, y la conclusión termina siendo los datos de salida.

En un proceso de cómputo todas las conclusiones deben ser verdaderas y correctas. En nuestro ejemplo, las formas de corregir el problema son *a* proporcionando al proceso elementos para que sea autosuficiente, o proporcionar más datos de entrada. Esto es una regla general del procesamiento: la parte del proceso que no pueda cubrirse con procesamiento deberá ser cubierto con datos de entrada.

Veamos cómo sería el ejemplo si cubrimos la falta de datos con procesamiento.

ilo is o

El área es igual a base por altura entre dos y la altura es la mitad de la base (V)

La base es 10 (V)

Δ El área es 25 (V) (C)

La forma de cubrir la deficiencia del proceso con datos de entrada es como se muestra en el silogismo A.

Lo que se puede desprender de esta explicación es que siempre debemos garantizar que los datos de salida sean verdaderos y correctos. Para ello es necesario verificar si el proceso es capaz de producir, a partir de los datos de entrada, los datos de salida. En caso de que no sea así, es necesario que se modifique el proceso o que se proporcionen los datos faltantes.

Finalidad de los enunciados en el proceso

Un proceso puede llegar a ser un poco complicado, por lo que es posible dividir las premisas de tal forma que nos den un marco de referencia más claro. Vea cómo en el Silogismo C el primer enunciado tiene dos juicios de verdad aplicables en nuestro programa. Podemos dividirlos para simplificar, dado que son independientes entre sí.

ilo is o

El área es igual a base por altura entre dos (V)

La altura es la mitad de la base (V)

La base es 10 (V)

Δ El área es 25 (V) (C)

Cuando trabajamos con la lógica de un programa es necesario dividir el silogismo en tantos enunciados como se pueda, siempre y cuando no sean interdependientes. Se conoce como *finalidad del enunciado* a la participación de los enunciados dentro de un programa de cómputo.

La finalidad de los enunciados pude identificarse de la siguiente manera.

1. **Enunciados de asignación de valores.** Son los enunciados que dan a conocer datos cuyo valor puede ser variable. Este tipo de enunciados nos lleva generalmente a la recopilación del dato, ya sea preguntando el dato al usuario, o leyéndolo de un medio de almacenamiento permanente. Típicos ejemplos serían los siguientes.

    a. La base es igual a 20.

    b. La altura es igual a 10.

    c. El coche es rojo.

    d. El saldo insoluto es de 230,000 pesos.

    e. La tasa de interés es de 3 mensual.

2. **Enunciados de valor preexistente.** Son los enunciados que dan a conocer datos cuyo valor es generalmente aceptado, y que no es necesario preguntar. Típicos ejemplos serían los siguientes.

    a. 60 minutos son equivalentes a una hora.

    b. El valor de PI es 3.1416 .

3. **Enunciados de cálculo.** Son aquellos que involucran una operación entre datos conocidos. Típicos ejemplos serían los siguientes.

    a. Los intereses sobre saldos insolutos se calculan aplicando la tasa de interés al saldo insoluto.

    b. El área de un triángulo rectángulo se obtiene multiplicando la base por la altura, y dividiendo el resultado entre dos.

4. **Enunciados condicionales.** Son aquellos que provocan un cálculo o asignación dependiendo de un juicio de valor. Típicos ejemplos serían los siguientes.

    a. Si la base es mayor a cero, entonces la altura es igual a la base entre dos.

    b. Si la existencia es mayor que la cantidad requerida, entonces nueva existencia es igual a existencia menos cantidad requerida.

. **Enunciado de resultado.** Son aquellos enunciados de asignación de valores, producto de la aplicación de los enunciados de cálculo. Típicos ejemplos serían los siguientes.

    a. El interés sobre saldos insolutos es de 6,900 pesos.

    b. El área del triángulo rectángulo es de 100.

En un programa lo más común es que se recopile información (enunciado de asignación de valores), que se complementen con reglas generales (valores preexistentes), y se procesen (enunciados de cálculo) para generar datos de salida (enunciado de resultado). Es posible que un proceso sólo utilice valores

preexistentes, o que sólo utilice valores asignados, o la mezcla de ambos tipos de valores.

En ocasiones los enunciados de resultados no son los resultados finales del programa; es muy probable que un enunciado de resultado provea de información a un enunciado de cálculo, dando lugar a las dependencias.

Si analizamos el Silogismo D nos daremos cuenta que los primeros dos enunciados son de cálculo, el tercero es enunciado de asignación de valores, y el último es un enunciado de resultado.

Dos enunciados son *enunciados dependientes* si la única forma de resolver el valor de un enunciado es habiendo resuelto primero el valor de otro enunciado. En nuestro ejemplo el segundo enunciado es dependiente del primero, dado que si no conocemos la altura no podemos conocer cuál es la mitad de la altura.

Principios en la elaboración de enunciados

Cada enunciado de un silogismo representa para un proceso datos y procesamiento. A fin de que nuestros programas sean eficientes y manejables, es necesario respetar algunos principios básicos.

Principios relacionados con los datos de entrada

Los siguientes principios son aplicables a los datos de entrada.

- **Principio de economía de los datos.** Se debe proveer a un proceso del mínimo de datos suficientes y necesarios para su ejecución. Si se proporcionan menos datos de los necesarios, el proceso no podrá realizarse; si se proporcionan datos de más, el proceso tendrá una carga adicional de información que tendrá que discriminar en el proceso.
- **Principio de autosuficiencia de los datos.** Los datos que pueden ser calculados a partir de otros datos de entrada, deben omitirse como datos de entrada. Proporcionar como datos de entrada la fecha de nacimiento y la edad de una persona es redundante: si conocemos la fecha de nacimiento podemos calcular la edad.
- **Principio de compatibilidad de los datos.** Los datos proporcionados deben ser del tipo de dato que sea compatible y fácil de manejar por el proceso. El precio de un artículo requerido para el cálculo en una factura puede ser proporcionado como 12,000, o como “doce mil”. Sin duda alguna, el número en su representación textual no nos servirá para realizar operaciones aritméticas, por lo que no tendría compatibilidad.
- **Principio de conocimiento preestablecido.** Si hay datos que son verdades generalmente aceptadas y que no van a cambiar, evite preguntarlas, ya que forman parte del proceso. Sería ocioso preguntar cuántos minutos componen una hora, dado que está generalmente aceptado que son 60.

Principios relacionados con el proceso

Los siguientes principios son aplicables al proceso.

- **Principio de consistencia del resultado.** En un proceso dado, ante iguales datos de entrada, igual deberá ser el resultado.
- **Principio de eficiencia del proceso.** En un proceso dado, debe obtenerse la mayor cantidad de resultado utilizando la menor cantidad de datos de entrada.
- **Principio de eficacia del proceso.** En un proceso dado, el proceso debe hacer exactamente lo que se requiere que haga. No debe hacer ni más ni menos.
- **Principio de simplicidad del proceso.** En un proceso dado, es preferible un conjunto compacto y sencillo de operaciones, que uno largo y complejo.
- **Principio de la independencia de los procesos.** En un proceso dado, debe evitarse dependencias con elementos externos al proceso, cuando éstos constituyan en sí mismos un proceso con una finalidad distinta a la reutilización.
- **Principio de reutilización de los procesos.** En un proceso dado, debe procurarse la utilización de otros procesos ya existentes, cuya finalidad es brindar funcionalidad a otros procesos.
- **Principio de rastreabilidad de las operaciones.** Todas las operaciones de un proceso deben poderse rastrear y auditar. No deben existir en los procesos “cajas negras”, en donde nadie sabe qué sucede en el proceso.

Principios relacionados con los datos de salida

Los siguientes principios son aplicables a los datos de salida.

- **Principio de cantidad y forma.** Los datos de salida deben generarse en la cantidad y forma requeridos por el usuario.
- **Principio de exactitud.** Los datos de salida deben tener el grado de exactitud requerido por el usuario.
- **Principio de dominio de salida.** Los datos de salida pueden tener diferente dominio que los datos de entrada, y pueden no ser compatibles entre sí. Por ejemplo, dividir dos números enteros no siempre da un número entero.

Ejercicio 03.02

Analizando un silogismo y clasificando las premisas

En el ejercicio 01.01 se calculó la cantidad de tiempo que se requiere para descargar una película desde Internet, utilizando una conexión de tipo Dial Up.

A continuación se muestran un conjunto de enunciados relacionados al problema. Analícelos y clasifíquelos adecuadamente.

En la columna Falso o Verdadero, coloque una V si el enunciado es verdadero, y coloque una F si el enunciado es falso.

En la columna Finalidad del enunciado, escriba si el enunciado es un enunciado de asignación (A), de valor preexistente (V), de cálculo (CA), condicional (CD) o de resultado (R).

Enunciado	Falso o verdadero	Finalidad del enunciado
1. La velocidad del módem es de 56 bits por segundo.		
2. 56 bits son 1024 bits.		
3. La tasa de transmisión es el 40 de la velocidad del módem.		
4. El 40 de 1024 bits son 409.6 bits.		
. La tasa de transmisión es de 409.6 bits por segundo.		
6. Un byte se forma de 8 bits.		
. Un GB es igual a 1073741824 bytes.		
. Un GB es igual a 8589934592 bits.		
. 40 GB son 343597383680 bits.		
10. Un MB es igual a 1073741824 bytes.		

11. Un GB es igual a 1024 bytes.		
12. Se requieren 838860800 segundos para transmitir 40 GB.		
13. 60 segundos equivalen a un minuto.		
14. 60 minutos equivalen a una hora.		
1 . Se requieren 233016.88 horas para transmitir 40 GB.		

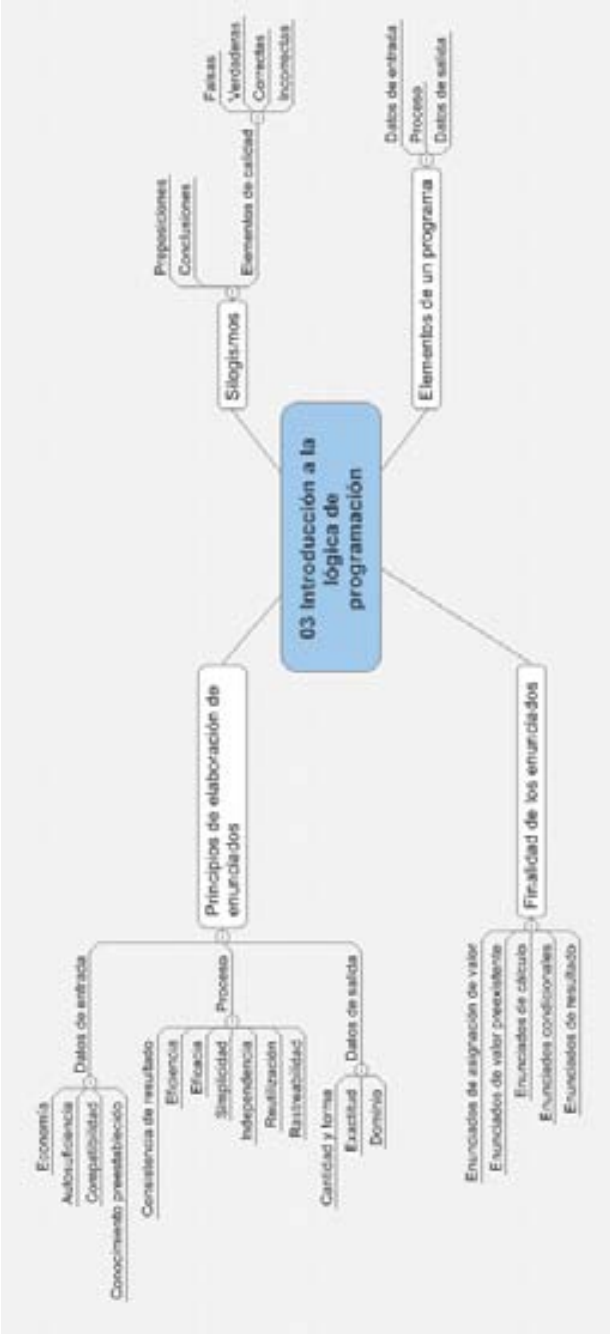
1. ¿Hay algún enunciado que no se utilice en el programa?

2. Si sustituimos en el programa los enunciados 13 y 14 por el siguiente enunciado: “3,600 segundos equivalen a una hora”, ¿qué principio de elaboración de enunciados estaríamos aplicando?

3. Aplicando el mismo principio que aplicamos en el punto anterior, ¿qué enunciados podemos eliminar?

FIN DEL EJERCICIO ✱

Mapa mental del capítulo



Terminología

Argumento, 37
Conclusión, 38
Dato de entrada, 42
Datos de salida, 42
Enunciado, 37
Enunciados declarativos, 38
Enunciados dependientes, 46
Finalidad del enunciado, 44
Lógica de programación, 42
Lógica, 37
Premisa mayor, 38
Premisa menor, 38
Premisas, 38
Procesar, 37
Proceso de razonar, 37
Proceso, 42
Programa de cómputo, 43
Proposición, 37
Razonar, 37
Silogismo, 37
Valor de verdad, 38

PREGUNTAS

3.1 ¿Considera usted posible que dos enunciados falsos den lugar a una conclusión verdadera?

3.2 A un programador le encargan un programa. Tratando de impresionar al usuario, el programador agrega funciones útiles que no fueron requeridas por el usuario inicialmente. Tomando en cuenta los principios de elaboración de enunciados ¿Es correcto lo que hizo el programador? Explique su respuesta.

3.3 En ocasiones el principio de independencia de procesos se opone al principio de reutilización de procesos. ¿Cuál es el criterio que adoptaría en caso de tener que decidir por cumplir uno u otro? Explique su respuesta.

3.4 Podemos considerar sinónimos a los argumentos y los silogismos.

3.5 El principio de eficiencia del proceso nos lleva a utilizar el máximo de valores preexistentes que podamos. ¿Es eso cierto? Explique su respuesta.

Notas:

Examen rápido

Parte 1 : Marque co n una cruz el inciso que considere que responde a la pregunta.

1) Expresión ling ística que establece un pensamiento completo.
(a) Enunciado
(b) Silogismo
(c) Conclusión

2) Dependiendo su apego con la realidad, las premisas pueden ser...
(a) Correctas e incorrectas
(b) Falsas y verdaderas
(c) Dependientes e independientes

3) Considerando el grado en que una conclusión se puede inferir de las premisas, la conclusión puede ser...
(a) Correcta o incorrecta
(b) Falsa o verdadera
(c) Dependiente o independiente

4) Es la habilidad de pensar de manera razonada, sistemática y ordenada, que nos hace capaces de inferir algoritmos abstractos para su implementación mediante programas de cómputo.
(a) Lógica
(b) Lógica de programación
(c) Manejo de principios

5) Son los enunciados que involucran una operación entre datos conocidos.
(a) Enunciados condicionales
(b) Enunciados de resultado
(c) Enunciados de cálculo

Parte 2: Coloque en la línea la letra «F» si la sentencia es falsa, y «V» si la sentencia es verdadera.

_____ **6)** Los enunciados de resultados siempre requieren la participación de enunciados de asignación de valores y enunciados de cálculo.

_____ **7)** Puede haber un silogismo sin premisas.

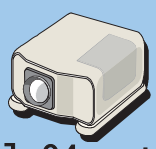
_____ **8)** El principio de exactitud nos obliga a buscar la exactitud de los resultados, aún y cuando el usuario sólo necesite resultados aproximados.

_____ **9)** El orden en que aparecen las premisas en un silogismo es importante, sobre todo cuando se tienen enunciados dependientes.

_____ **10)** Los datos de salida son producto de aplicar operaciones aritméticas a los datos de entrada.

4

La naturaleza de los datos



lp04.ppt

Objetivos: Aprenderá las particularidades de los datos como elementos de procesamiento y almacenamiento, sus límites y sus representaciones abstractas.

1. Aprenderá la naturaleza de los valores que representan datos para la computadora.
2. Aprenderá las propiedades de los datos.
3. Aprenderá el concepto de expresiones y variables.
4. Aprenderá los tipos de datos soportados por los lenguajes más actuales.
5. Aprenderá cuáles son los tipos de dominio.
6. Aprenderá a representar de forma abstracta a los datos y sus dominios.

www.detodoprogramacion.com

Contenido

Naturaleza de los valores 57
Propiedades de los datos 58
Expresiones y variables 59
Expresiones 59
Variables 59
Arreglos 60
Análisis de los nombres de variables 61
Los tipos de datos y su soporte en los lenguajes 62
Visual Basic.NET 62
C# 63
C++ 63
Java 64
Tipos de datos base 64
Dominios 65
Dominio de tipo 65
Dominio de regla de negocio 65
Dominio de relación 65
Representación abstracta de tipos base 66
Tipos de datos 66
Dominios 66
Identificando los tipos de datos idóneos para los datos 68
Análisis de economía de datos 69
Representación abstracta de dominios 70
Mapa mental del capítulo 71
Terminología 72
Preguntas 72
Examen rápido 73

i los datos no sirven la información no sirve.

Existe un principio denominado que significa "Basura entra, basura sale" (por sus siglas en inglés, *arbage n arbage ut*). Este principio indica que si a los programas les proporcionamos datos que no sirven, la información no servirá.

Es responsabilidad de los programadores hacer todo lo posible para que el programa no permita el manejo de datos equivocados. En cierta forma, la gran parte del trabajo de programación consiste en realizar una recopilación adecuada de los datos que desean procesar o almacenar.

Algunos otros libros que tratan los elementos de la programación comienzan a explicar el tema de los algoritmos sin haber hecho referencia a los datos; tratan una secuencia de actividades a detalle, en forma descriptiva. El resultado es lógico, dado

que pasan a ejemplos de algoritmos sin aterrizarlos en el mundo de los datos, dichos algoritmos no sirven para su implementación en programas de cómputo. En este libro le damos a los datos la relevancia que merecen, explicándolos a detalle.

Me arriesgo a emitir la siguiente máxima: en términos computacionales, si no se puede traducir a datos, no se puede automatizar.

Naturaleza de los valores

En términos computacionales, los valores que pueden ser manejados tienen las siguientes naturalezas.

- *alores num ricos.*

  Son todos aquellos valores con los cuales se pueden realizar cálculos aritméticos. Dentro de los valores numéricos tenemos diferentes tipos: los números enteros, que son aquellos que no poseen decimales; tenemos los números decimales, que son aquellos que sí poseen decimales; estos a su vez pueden tener precisión simple o doble, dependiendo del número de decimales que soportan. Estos números decimales también varían en cuanto a su exactitud, ya que hay números decimales que ofrecen exactitud completa, mientras que otros ofrecen resultados aproximados. El consumo de bytes para almacenar los valores numéricos depende del tipo de valor numérico y la exactitud requerida. Los tipos de valores numéricos son de longitud fija.

- *alores cadena.*

  Son todos aquellos valores compuestos por una serie de caracteres, que son tratados como un solo valor. Internamente, la computadora ve a cada símbolo de una cadena en forma independiente, sin embargo, los valores de tipo cadena permiten ver al conjunto de caracteres como un solo valor (“h” ”0” ”l” ”a” “hola”). Los valores de tipo cadena tienen longitud variable.

- *alores fecha.*

  Son todos aquellos valores que almacenan una fecha de calendario. Internamente para la computadora son números que representan de manera única a cada día calendario. Para ello se utilizan esquemas estándar, como por ejemplo las fechas julianas (*ulian ay umber*). Este tipo de valores permiten la comparación de fechas y los cálculos basados en tiempo, a través de representaciones de fecha reconocidas por nosotros (dd mm aa hh:mm:ss:mmm). Por ser en el fondo un número, los valores fecha son de longitud fija.

- *alores booleanos.*

  Son valores binarios compuestos por el conjunto 0, 1 , llamados así por tener sus orígenes en el álgebra de Boole. Este tipo de valores permiten manejar pares de valores mutuamente excluyentes, como Falso Verdadero,

Si No, Abierto Cerrado, etcétera. Como generalmente se utilizan para representar valores Falso Verdadero, suelen conocerse como *valores lógicos*.

- *alores nulos.*

  Es la ausencia de valor. Indica que el valor se desconoce; cualquier operación realizada con valores nulos genera valores nulos, dado que es imposible determinar el resultado final. Por ejemplo, en la siguiente operación (x = 4 + null), sería null; dado que, como no sabemos el segundo número de la suma, no podemos saber qué valor asumirá x.

- *alores .*

  Son valores compuestos por la representación binaria (digital) de grandes cantidades de información, tal como imágenes, sonidos, programas, etcétera. BLOB significa objetos binarios largos, por sus siglas en inglés (BLOB *inary arge bject*).

Con los valores es posible realizar operaciones aritméticas, lógicas y de comparación.

Propiedades de los datos

Los datos, como sabemos, son representaciones simbólicas no significativas. A través del procesamiento y la interpretación se convertirán en información; de ahí su importancia.

Dentro de un programa o algoritmo, los datos siempre tienen las siguientes propiedades.

- *ipo de dato.*

  Es un dominio nominado de datos que permite el almacenamiento de una determinada naturaleza de dato, atendiendo un grado de exactitud específico, causando un consumo determinado de almacenamiento.

- *ominio.*

  Es el conjunto de valores válidos para un dato.

- *pcionalidad.*

  Es la necesidad que se tiene de que un dato tenga valor.

- *alor.*

  Es la expresión almacenada como dato en un momento determinado.

Imagine que para un cálculo es necesario conocer su edad, expresada en años. Sin duda alguna lo que se necesita es un número entero (tipo de dato); dado el valor que se desea almacenar, un dato de tipo fecha o un conjunto de caracteres no nos servirían. Además sabemos que en su calidad de humano la edad en años que tiene,

puede estar entre 0 y 120 aproximadamente (dominio); un valor negativo, como podría ser 5 sería inadmisible. Sabemos también que el dato es necesario. Cumplidas todas las propiedades, sólo resta que la edad que tenga que proporcionar cumpla con todas las restricciones que marcamos.

Expresiones y variables

Expresiones

En términos de programación, una *e presión* es el valor de un dato.

Cualquier dato u operación que resulte en un dato es una expresión. Se trata de una *e presión absoluta* si su representación es literal (se escribe directamente), y se trata de una *e presión compuesta* si es necesario resolver un valor a partir de otros, o como resultado de aplicar una función sobre un dato.

Algunos ejemplos de expresiones son los siguientes.

1. Expresión absoluta de tipo numérico: 4.
2. Expresión absoluta de tipo cadena: “Dato”.
3. Expresión absoluta de tipo booleano: False.
4. Expresión compuesta de tipo numérico: 4 10 (la expresión vale 14).
. Expresión compuesta de tipo booleano: 4 3 (la expresión vale False).

Variables

Una *variable* es la referencia lógica nominada a la posición física de memoria RAM en donde se almacena una expresión.

Para poder utilizar una expresión en un programa, primero debe alojarse en memoria RAM; la única forma de colocar expresiones en la RAM es a través de las variables de memoria. Reciben este nombre dado que el valor puede cambiar durante el transcurso de la ejecución del programa.

Las variables son a fin de cuentas un nombre a través del cual podemos referir una posición en memoria. Los nombres de variables, por regla general, deben cumplir con las siguientes reglas:

1. Siempre deben iniciar con una letra.
2. Deben componerse de letras y dígitos.
3. No pueden contener espacios en blanco ni caracteres especiales.
4. Las letras acentuadas y otros signos regionales se consideran caracteres especiales.

. No deben exceder de 32 caracteres.

6. Deben ser representativos del valor que almacenan.

Estas reglas no son obligatorias para todos los lenguajes, sólo recomendables. Por ejemplo, los nombres pueden iniciar con guión bajo y también pueden exceder de 32 caracteres; sin embargo, si respetamos las reglas anteriores no tendremos problemas en ningún lenguaje.

En el caso de desarrollos en donde participa más de un programador es indispensable que se establezcan reglas de nomenclatura comunes para las variables, entendidas y respetadas por todos; esto permite que todos los miembros del equipo de desarrollo eviten el caos en el código y la discrecionalidad, lo cual siempre quita tiempo al momento de analizar porciones de código. Los programadores generalmente están sometidos a mucha presión, y en un ambiente de stress, no es raro encontrarnos nombres de variables fuera de lugar, tales como el nombre de la mascota o de un grupo musical de nuestra preferencia. Esas prácticas, en el ambiente profesional, deben evitarse.

Arreglos

En ocasiones existen conjuntos de datos que son muy similares en su naturaleza y valor almacenado, por lo que ameritan ser considerados una misma cosa.

Por ejemplo, si queremos almacenar el monto de las ventas de cada uno de los trimestres del año, podríamos pretender almacenar la información en un conjunto de variables que bien podrían llamarse así.

Trimestre1

Trimestre2

Trimestre3

Trimestre4

En este caso se trata de 4 variables, por lo que no es problema. La situación se complicaría si habláramos de los montos de las ventas de los trimestres de los últimos 20 años. Tendríamos que definir 80 nombres de variables distintos. Más se complicaría si nos piden sumar todos los montos, ya que no podríamos automatizar nada.

Los *arreglos* son variables que permiten asociar, a través de un mismo nombre de variables, varias posiciones de memoria.

Esto es muy útil cuando se desea almacenar un conjunto de datos de una misma naturaleza, sin tener que recurrir a nombres diversos de variables.

Aunque unidas por un mismo nombre, las posiciones de memoria manejadas en forma de arreglo se diferencian por un número único al que conocemos como

sub ndice. El subíndice es un número entero no negativo. Dependiendo del lenguaje, puede iniciar desde cero o desde uno.

A cada posición de memoria almacenada en el arreglo se le llama *elemento* (*item*).

Volviendo a nuestro ejemplo, podríamos definir un arreglo como sigue.

Trimestre(*n*)

Donde *n* sería un subíndice que podría valer del 1 al 4, indicando el número de trimestre del que se trate.

Los arreglos pueden ser también de dos dimensiones, es decir, que pueden almacenar datos de forma bidimensional. En ese caso, se debe especificar el subíndice indicando la fila y columna del dato que deseamos.

Trimestre(*n m*)

Donde *n* sería el subíndice que indica la fila, y *m* el subíndice que indique la columna.

Ejercicio 04.01

Análisis de los nombres de variables

Escriba en la línea de la derecha una letra "i" si el nombre de la variable es incorrecto, y "c" si es correcto. Tome en cuenta las reglas sugeridas para la nomenclatura de variables.

Nombre _____

Ventas Totales _____

VentasTotales _____

Ventas Totales _____

Ventas Torales _____

01Inicial _____

Ingresos Egresos _____

NombrePersonaEncargadaDelAnalisis DiseñoDeSistemas

MétodoUtilizado _____

IngresosEgresosDelEjercicio _____

FIN DEL EJERCICIO ✱

Los tipos de datos y su soporte en los lenguajes

Un *tipo de dato* es un dominio nominado de datos que permite el almacenamiento de una determinada naturaleza de dato, atendiendo un grado de exactitud específico, causando un consumo determinado de almacenamiento.

Cada lenguaje soporta un número determinado de tipos de datos, a través de los cuales manejan las diferentes naturalezas de datos que existen.

Para efectos de este libro estableceremos tipos básicos que nos permitan demostrar el desarrollo de algoritmos; no es nuestra intención entrar a fondo respecto a los tipos de datos y la forma en que se comportan en el uso del lenguaje.

A continuación se describen los tipos de datos generales que soportan los lenguajes VB.NET, C , C y Java, en las versiones tratadas en este libro; aunque los lenguajes poseen más tipos de datos que los relacionados, no se mencionan más que los que utilizaremos en los ejercicios del libro. Le dimos énfasis a los datos de tipo numérico, dado que los datos más complejos, como las fechas y las cadenas, tienen diversas formas de implementación que dificultarían el aprendizaje de los algoritmos, que es nuestro tema de interés.

Visual Basic.NET

Estos son los tipos de datos básicos en Visual Basic.NET.

Tipo	Consumo en Bytes	Dominio de tipo
Boolean		r e o alse
Byte	1 byte	0 a 255 (sin signo).
Decimal	16 bytes	0 a 79.228.162.514.264.337.593.543.950.335 sin separador decimal; 0 a 7,9228162514264337593543950335 con 28 posiciones a la derecha del signo decimal; el número más pequeño distinto de cero es 0,0000000000000000000000000001 (1E 28).

Integer	4 bytes	2.147.483.648 a 2.147.483.647
String	Clase (2 bytes por caracter, aprox)	Cadenas de caracteres.

C#

Estos son los tipos de datos básicos en C

Tipo	Consumo en Bytes	Dominio de tipo
bool		r e o alse
byte	1 byte	0 a 255 (sin signo).
decimal	16 bytes	0 a 79.228.162.514.264.337.593.543.950.335 sin separador decimal; 0 a 7,922816251426433759354395033 5 con 28 posiciones a la derecha del signo decimal; el número más pequeño distinto de cero es 0,0000000000000000000000000001 (1E 28).
int	4 bytes	2.147.483.648 a 2.147.483.647
String	Clase (2 bytes por caracter, aprox)	Cadenas de caracteres.

C++

Estos son los tipos de datos en C

Tipo	Consumo en Bytes	Dominio de tipo
bool		r e o alse
short	2 bytes	32,767 a 32,767
double	8 bytes	De 10^{308} a 10^{308}

int	4 bytes	2.147.483.648 a 2.147.483.647
string	Clase (1 byte por caracter, aprox)	Arreglo de caracteres.

Java

Estos son los tipos de datos en Java

Tipo	Consumo en Bytes	Dominio de tipo
boolean		r e o alse
byte	1 byte	128 a 127
long	8 bytes	9,223,372,036,854,775,808 a 9,223,372,036,854,775,807
int	4 bytes	2,147,483,648 a 2,147,483,647
String	Clase (1 byte por caracter, aprox)	Arreglo de caracteres.

Tipos de datos base

Estos son los tipos de datos base para todos los ejercicios del libro.

Tipo común	VB.NET	C#	C++	Java
Booleano (l)	Boolean	bool	bool	boolean
Byte (b)	Byte	byte	short	byte
Cadena (x)	String	String	string	String
Decimal (d)	Decimal	decimal	double	long
Entero (i)	Integer	int	int	int

No en todos los casos se tienen tipos de datos totalmente equivalentes, por lo cual las equivalencias son sólo aproximadas, con el afán de uniformar las diferencias entre lenguajes.

Dominios

El tema de los dominios es muy importante. Su definición clara permite evitar el ingreso de datos equivocados a los procesos o al almacenamiento. Como ya se dijo, el dominio de un dato es el conjunto de valores válidos que es posible que el dato adquiera. A la cantidad de valores permitidos en un dominio se le llama *amplitud de dominio*.

Se tienen tres tipos de dominio: dominio de tipo, dominio de regla de negocio y dominio de relación.

Dominio de tipo

El *dominio de tipo* es la restricción de valores originada por el tipo de dato especificado. Por ejemplo, sabemos que el tipo de dato byte sólo permite almacenar números enteros entre 0 y 255; cualquier dato definido con un determinado tipo de dato no podrá exceder sus límites. Por ejemplo, en un dato byte no sería posible almacenar el número 1,000, dado que 1,000 está más allá de los límites del tipo.

Dominio de regla de negocio

El *dominio de regla de negocio* es la restricción de valores originada por una regla específica del negocio que restringe la amplitud de dominio; aún y cuando la amplitud de dominio del tipo permita muchos valores, la regla del negocio puede reducir dicha amplitud en base a supuestos o restricciones que se deben respetar. Un ejemplo podría ser un dato relacionado con la edad de un empleado. Aunque un tipo de dato byte es suficiente para almacenar la edad de cualquier persona (0 a 255), la empresa puede tener una regla de negocio que dice que los empleados de la compañía sólo podrán tener entre 18 y 65 años; de esa forma, aunque el tipo de dato permita el valor de 200, la regla de negocio no la permite.

Dominio de relación

El *dominio de relación* es la restricción de valores originada por la relación que el dato debe mantener con otros datos. Este tipo de dominio es común cuando se tienen fechas iniciales y finales de un período. La fecha final debe ser mayor o igual que la fecha inicial. En ese sentido, el dominio de la fecha final está determinado por el valor que adquiera la fecha inicial. También es común ver este tipo de dominio al utilizar bases de datos. Suponga que la matrícula en una institución educativa se compone de un número entero (dominio de tipo), que por regla general deberá estar entre el 100,000 y el 10,000,000 (regla de negocio); aún y con estas restricciones, un alumno no puede utilizar varios números de matrícula para identificarse: es necesario que utilice el número de matrícula específica que le fue asignado. Muchos datos en una base de datos deben estar registrados antes de poder ser utilizados, como es el caso de códigos de productos, números de empleados, números de seguro social, etcétera. Para el primer registro que se hace de los datos sólo aplicarían los dominios de tipo y de regla de negocio, pero a partir de ahí, cada vez que el dato se invoque deberá cumplirse con el dominio de relación.

www.detodoprogramacion.com

Representación abstracta de tipos base

Tipos de datos

Para efectos de este libro, utilizaremos este cuadro básico de datos, dado que son posibles de representar en todos los lenguajes.

Tipo común	Símbolo Abstracto
Booleano	l
Byte	b
Cadena	x(*n*) Donde *n* es la longitud en caracteres de la cadena.
Decimal	d
Entero	i
Fecha	f

Por mucho, el tipo de dato que más eficientemente es manejado por las computadoras con arquitecturas de 32 bits (que son las más comunes hoy en día) son los datos enteros, dado que se representan en 32 bits y no requieren trabajos adicionales para determinar su valor.

En representación abstracta, deberá incluirse el símbolo abstracto del tipo antes de la representación del dominio. El símbolo abstracto se encuentra en la tabla de tipos base. En el caso de los datos de tipo cadena, se debe especificar antes del dominio la longitud esperada para el valor, encerrándolo en paréntesis.

Dominios

Conjunto universo y subconjuntos. Al analizar todas las restricciones de dominio de los datos, la resultante es siempre un conjunto de valores. Considerando ello, podremos representar el dominio de un dato encerrando la serie de valores entre corchetes “ ”, utilizados generalmente para la representación de un *conjunto universo* (todos los valores). Si se tienen *subconjuntos* en el conjunto universo, éstos se representarán utilizando llaves “ ”.

Series continuas y discontinuas. El conjunto de valores puede ser una *serie continua de valores*, si el conjunto incluye todos los valores permitidos por un tipo en un límite dado; en ese caso, la representación podrá realizarse separando los valores límite por un guión medio “ “.

El conjunto de valores puede ser una *serie discontinua de valores*, si el conjunto está formado sólo por algunos valores dentro de los valores permitidos en un límite dado. En este caso, los valores permitidos se deberán separar por comas “, “. Para efectos de representación, “n” representa el límite máximo permitido por un tipo de datos.

En caso de que se tenga más de un conjunto, todos los conjuntos deberán especificarse entre llaves.

Se proponen los siguientes ejemplos de representación.

Dominio de tipo para los datos Byte.	b 0 255
Dominio para un entero, dónde sólo se permiten los números de 100,000 al 200,000.	i 100000 200000
Dominio para un dato Byte, en donde sólo se permiten los datos 3, 5 y 7.	b 3,5,7
Dominio de datos Byte, en donde sólo se permiten del 1 al 100 y del 150 al 200.	b 1 100 , 150 200
Una cadena en donde sólo se permiten letras en mayúsculas, dígitos y espacios en blanco (*lan paces*).	x(40) A , 0 9 , BS
Un número entero, de 5,000 en adelante.	i 5000 n

Series lógicas de valores. Se tiene una *serie lógica de valores* cuando un conjunto de valores obedece a una regla aritmética evidente. En este caso, podrán utilizarse puntos continuos “...” para señalar que la serie sigue hasta un límite dado. Si se especifica “n”, quiere decir que la serie continúa hasta alcanzar el límite máximo que permita el tipo.

Se proponen los siguientes ejemplos de representación.

La serie del 5.	b 5,10,15,...,50
Todos los múltiplos de 10 posibles en un dato entero.	i 10,20,30,...,*n*

Exclusiones. Estamos ante la presencia de *e clusiones* cuando sólo un subconjunto de valores dentro del conjunto no puede ser elegido. Para ello utilizamos el símbolo de exclusión negación, que es el símbolo de admiración “ ”, antes de especificar el subconjunto a ser excluido.

Todos los números Byte, excepto el cero.	b 0
Los números Byte del 1 al 100, excepto el 25, 50 y 75.	b 1 100 , 25,50,75

Reglas de negocio. La abstracción de las *reglas de negocios* es más compleja, dado que en ocasiones son procesos completos que ameritan una representación más formal y extendida, como los diagramas de flujo.

Posiciones. En ocasiones un subconjunto está representado por una determinada cantidad de caracteres específicos. En ese caso, es necesario especificar el número de *posiciones* que un determinado subconjunto puede representar de una expresión. La forma de representar las posiciones es anteponiendo el número de caracteres que estará sujeto a un patrón.

Una clave compuesta por 4 letras y 6 números.	x(10) 4 A ,6 0 9
Un teléfono en formato 99 (99) 9999 9999.	x(17) 2 0 9 (2 0 9) 4 0 9 4 0 9

Ejercicio 04.02

Identificando los tipos de datos idóneos para los datos

Considerando los datos que se proponen, especifique qué tipo de dato les aplicaría. Debe elegir el tipo de dato que pueda almacenar el dato que se propone, con el mínimo consumo de almacenamiento.

1. Se desea almacenar la edad de una persona, en Visual Basic.NET.

2. Se desea almacenar el nombre de una persona, que no excede de 40 caracteres, en Java.

3. Se desea almacenar la distancia en centímetros que existe de la Tierra a Venus.

FIN DEL EJERCICIO ✱

Ejercicio 04.03

Análisis de economía de datos

Un programador utilizó los siguientes tipos de datos para automatizar un archivo de empleados de una pequeña empresa que tiene 45 empleados. Analice los datos que utilizó y haga usted una propuesta que consuma menos recursos de almacenamiento en Bytes.

Número de empleado.	i 1 45
Nombre del empleado.	x(200)
Edad.	i
Clave del puesto . (se tienen 5 diferentes puestos en la compañía, y el programador decidió ponerles un número de identificación, aunque eso no es obligatorio).	i
Sueldo mensual en pesos y centavos. (Los sueldos van de 5,540.45 a 28,380.70).	d

FIN DEL EJERCICIO ✱

Ejercicio 04.04

Representación abstracta de dominios

Considerando las formas de representación abstracta de dominios, escriba la representación aplicable a los siguientes casos.

1. Un número entero.

2. Un número entero positivo.

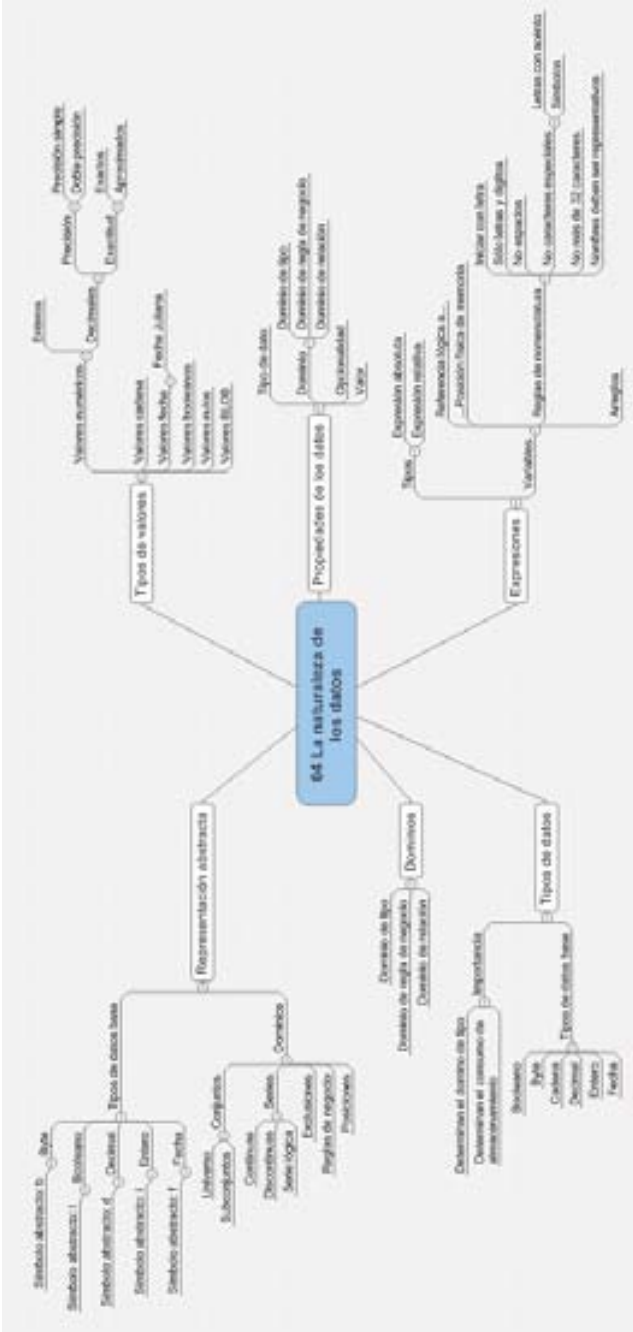
3. Una clave formada por 4 dígitos y 5 letras.

4. Un número Byte entre 16 y 65.

. Una clave formada por una letra “A” y 4 números. El primero de los 4 números no podrá ser cero.

FIN DEL EJERCICIO ✱

Mapa mental del capítulo



Terminología

Amplitud de dominio, 65
Arreglos, 60
Binary Large Object, 58
BLOB, 58
Conjunto universo, 66
Dominio de regla de negocio, 65
Dominio de relación, 65
Dominio de tipo, 65
Dominio, 58
Elemento, 61
Exclusiones, 67
Expresión absoluta, 59
Expresión compuesta, 59
Expresión, 59
Garbage In, Garbage Out, 56
GIGO, 56
Item, 61
JDN, 57
Julian Day Number, 57
Opcionalidad, 58
Posiciones, 68
Reglas de negocios, 68
Serie continua de valores, 66
Serie discontinua de valores, 67
Serie lógica de valores, 67
Subconjuntos, 66
Subíndice, 61
Tipo de dato, 62
Valor, 58
Valores BLOB, 58
Valores booleanos, 57
Valores cadena, 57
Valores fecha, 57
Valores lógicos, 58
Valores nulos, 58
Valores numéricos, 57
Variable, 59

Preguntas

4.1 Si su equipo posee un disco duro de altísima capacidad, ¿por qué sigue siendo importante cuidar el consumo en bytes de los datos que utilizan nuestros programas?

4.2 Una empresa adquiere el sistema SAP, que es uno de los ERP más sofisticados y reconocidos a nivel mundial. La calidad del sistema es indiscutible, y ha probado ser útil para muchas empresas. Sin embargo, la empresa que acaba de comprar el sistema, después de 6 meses de uso, encuentra que los informes que está arrojando el sistema están equivocados y contienen información obsoleta. ¿Llamaría a la empresa desarrolladora del soft are para decirles que el programa tiene errores, dado que está emitiendo información incorrecta?

4.3 ¿Qué pasa si un grupo de programadores que trabaja de manera conjunta en el desarrollo de una aplicación empresarial no se pone de acuerdo en las reglas de nomenclatura para las variables del sistema?

4.4 Existe alguna relación directa en el tipo de dato de un dato de entrada y uno de salida.

Examen rápido

Parte 1 : Marque co n una cruz el inciso que considere que responde a la pregunta.

1) Fenómeno que sostiene que si los datos de entrada son malos, los datos de salida son malos también.
(a) FIFO
(b) FODA
(c) GIGO

2) Son los únicos tipos de valores de longitud variable.
(a) Valores cadena
(b) Valores fecha
(c) Valores numéricos

3) Es la propiedad de los datos que determina la naturaleza y los límites preestablecidos de un dato.
(a) Dominio
(b) Opcionalidad
(c) Tipo de dato

4) Es la referencia lógica nominada a una posición de memoria física.
(a) Expresión
(b) Variable
(c) Dominio

5) Es el tipo de dominio que tiene que ver con otros datos
(a) Dominio de tipo
(b) Dominio de regla de negocio
(c) Dominio de relación

Parte 2: Coloque en la línea la letra «F» si la sentencia es falsa, y «V» si la sentencia es verdadera.

6) La suma de dos números enteros siempre da un número entero.

7) Es posible utilizar en un proceso datos sin cargarlos en memoria RAM, siempre y cuando utilicemos variables.

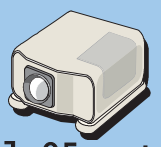
8) El tipo de dato que más eficientemente es manejado por la computadora es el tipo Byte, dado que es la unidad mínima de trabajo.

9) El símbolo “b” se utiliza para representar el tipo de dato Booleano.

10) El dominio de tipo es el que ofrece la mayor amplitud de tipo.

5

Operadores y reglas de precedencia



lp05.ppt

Objetivos: **Aprenderá cuáles son los operadores más comunes al tra bajar con los lenguajes de programación, y aprenderá sus reglas de precedencia.**

1. **Aprenderá cuáles son las categorías de operadores que existen.**
2. **Aprenderá cuáles son los ope radores aritmé ticos, de a signación, comparativos y lógicos, para la construcción de procesos computarizados.**
3. **Aprenderá en qué consisten las reglas de precedencia implícitas, posicionales y explícitas.**
4. **Aprenderá a construir expresiones complejas.**

www.detodoprogramacion.com

Contenido

Categorías de operadores 76
Operadores aritméticos 76
Operadores de asignación 78
Operadores comparativos 78
Operadores lógicos 79
Reglas de precedencia 80
Precedencia implícita 81
Precedencia posicional 82
Precedencia explícita 82
Aplicación de operadores y sus reglas de precedencia 83
Representación de expresiones complejas operadores y elementos de lógica simbólica 85
Mapa mental del capítulo 88
Terminología 89
Preguntas 89
Examen rápido 91

o ue no es verdad es mentira.

Mentira ← (¬ Verdad)

Los *operadores* son los elementos del lenguaje que nos permitirán asignar, calcular y comparar expresiones, dando lugar a lo que conocemos como procesamiento.

Todos los lenguajes poseen operadores, siendo las clasificaciones más importantes las siguientes: aritméticos, de asignación, de comparación y lógicos.

Categorías de operadores

Operadores aritméticos

Los operadores aritméticos forman parte de la educación básica de cualquier persona, por lo cual difícilmente podríamos agregar algo que no sepa con respecto a ellos y lo que representan. Si en su vida ya ha sumado, restado, multiplicado y dividido, en este libro no tenemos nada nuevo que explicarle.

No obstante lo obvio, para no ser omisos definiremos a los *operadores aritm ticos* como los símbolos que producen, a partir de dos expresiones numéricas conocidas,

una nueva expresión numérica, como resultado de la aplicación de un cálculo aritmético entre aquellas.

La mayoría de los lenguajes disponen de los siguientes operadores aritméticos:

Operador	Nombre	Función y sintaxis
+	Suma	Suma, expresiones numéricas. Resultado = Expresión1 + Expresión2
–	Resta	Representa la diferencia entre dos números o especifica la condición negativa de uno. Resultado = Número1 – Número2
*	Multiplicación	Multiplica dos números. Resultado = Número1 * Número2
/	División	Divide un número entre otro. Resultado = Número1 / Número2
^	Exponencial	Sirve para elevar un número a una determinada potencia; un número puede ser negativo si el exponente es un entero. Resultado = Número1 ^ Número2

La forma en que se resuelven este tipo de operadores es la siguiente:

Resultado = *Expresión1* `Operador` *Expresión2*

Expresión1 y *Expresión2* serán siempre datos numéricos. *Resultado* será siempre numérico.

Se debe tomar en cuenta que *Resultado* puede no ser del mismo tipo que las expresiones, por ejemplo, la suma de dos enteros (2,000,000,000 y 2,000,000,000) puede no ser entero.

Operadores de asignación

El *operador de asignación* es el que permite asignar un valor a una variable. El operador clásico de asignación es el signo de igual ( ).

Variable = *Valor*

Donde *Variable* es una variable y *Valor*, una expresión válida para el tipo de dato de *Variable*.

Desde el punto de vista de la lógica simbólica, la asignación se representa con el símbolo "←", que indica que una expresión pasa a una variable. Nuestra sintaxis original quedaría de la siguiente forma.

Variable ← Valor

Operadores comparativos

Los *operadores comparativos* son los que permiten comparar expresiones, una en relación a la otra, proporcionando un valor de falso (False), verdadero (True), o nulo (Null), dependiendo si la comparación es una verdad o no.

La tabla que sigue describe los operadores comparativos clásicos:

Operador	Verdadero si	Falso si
< (Menor que)	*e presión1* *e presión2*	*e presión1* *e presión2*
<= (Menor o igual que)	*e presión1* *e presión2*	*e presión1* *e presión2*
> (Mayor que)	*e presión1* *e presión2*	*e presión1* *e presión2*
>= (Mayor o igual que)	*e presión1* *e presión2*	*e presión1* *e presión2*
= (Igual a)	*e presión1* *e presión2*	*e presión1* *e presión2*
<> (Distinto de)	*e presión1* *e presión2*	*e presión1* *e presión2*

El resultado de la comparación será Null si cualquiera de las expresiones comparadas es Null.

La forma en que se resuelven este tipo de operadores es la siguiente:

Resultado = *Expresión1* `Operador` *Expresión2*

Expresión1 y *Expresión2* pueden ser cualquier tipo de expresiones comparables entre sí. *Resultado* será siempre lógico (True, False), o nulo (Null).

Algunos ejemplos del uso de estos operadores son:

"A" = "B"	retorna False
1 > 0.5	retorna True
2 >= 2	retorna True
"S" < "s"	retorna True

En el caso de los datos de tipo cadena, se comparan en base al código que tienen asignado en la página de caracteres. Tome en cuenta que las letras mayúsculas, para efectos de comparación, tienen un valor en código ASCII menor a las minúsculas, y por tanto, son menores. Siendo así, la "A" es menor que la " ", pero "z" es mayor a "A".

Operadores lógicos

Los *operadores lógicos* son aquellos que sirven para unir o negar condiciones, produciendo un valor lógico. Los operadores lógicos básicos son los siguientes.

Nombre	Operador	Comportamiento	Exp. Lógica simbólica
Negación	Not	Niega el resultado de una condición. Revierte el valor; si la condición que afecta es True producirá False, y viceversa.	¬
Disyunción	And	Cuando de entre dos condiciones, las dos deben ser True para que en su conjunto la expresión sea True.	∧
Conjunción	Or	Cuando de entre dos condiciones, al menos una debe ser True para que en su conjunto la expresión sea True.	∨

La forma en que se resuelven este tipo de operadores es la siguiente:

Resultado = [*Expresión1*] Operador *Expresión2*

Expresión1 y *Expresión2* son expresiones de tipo lógico. En el caso de Not, *Expresión1* no se debe poner, ya que el operador actúa sobre una sola expresión, que deberá ser lógica. *Resultado* será siempre lógico (True, False).

Cuando una expresión lógica se compone por una sola comparación, se dice que es una *e presión lógica simple*, debido a que sólo se resolverá en un tiempo (se resuelve la comparación y es todo); si involucra dos o más comparaciones, o la utilización de un operador lógico, se dice que es una *e presión lógica compuesta* debido a que la expresión lógica deberá resolverse en dos o más tiempos (se resuelve la comparación, y luego se resuelve dentro del contexto en que se encuentra). Una expresión lógica compuesta siempre se compone de expresiones lógicas simples, afectadas por operadores lógicos que las obligan a resolverse como una sola expresión lógica.

Ejemplos:

¬ True	Es False, porque es lo contrario a True
¬ False	Es True, porque es lo contrario a False
¬ "A" = "B"	Es True, porque "A" = "B" es False, pero negado es True
"A" = "B" ∧ 1 > 0.5	Es False, porque no todas las condiciones son True.
"A" = "B" ∨ 1 > 0.5	Es True, porque al menos una condición es True.

En caso de que tenga más de dos condiciones conjuntas, entra en operación lo que se conoce como *preferencia* que consiste en determinar el orden en que las condiciones u operaciones se han de resolver.

Reglas de precedencia

Podemos definir a la *precedencia* como la característica de una expresión compuesta de resolverse con anterioridad a otras.

Existen los siguientes tipos de precedencia: implícita, posicional y explícita.

Precedencia implícita

La *precedencia impl cita* es aquella inherente a los operadores y la categoría a la que pertenecen.

Precedencia im plícita p or cate goría. La *precedencia impl cita por categor a* se presenta cuando hay expresiones que contienen operadores de más de una categoría (aritméticos, comparativos y lógicos). Por regla general se resuelven antes las expresiones que involucran operadores aritméticos, a continuación se resuelven las expresiones que involucran operadores de comparación y por último se resuelven las expresiones que involucran operadores lógicos.

Ejemplo:

8 > 9 ∧ 4 + 3 > 5	Se resolverían primero las expresiones que involucran expresiones aritméticas.
8 > 9 ∧ 7 > 5	Luego se resolverían las expresiones que involucran expresiones comparativas.
False ∧ True	finalmente se resolvería la expresión que involucra operadores lógicos.
False	Expresión resuelta.

Precedencia implícita por operador. La *precedencia impl cita por operador* es la que tiene un operador respecto a los operadores de su misma categoría.

En el caso de los operadores de comparación, todos tienen la misma precedencia implícita por operador.

Los operadores aritméticos se evalúan en el siguiente orden de prioridad.

Aritméticos
Exponenciación
Negatividad
Multiplicación y división
Adición y substracción

Los operadores lógicos se evalúan en el siguiente orden de prioridad.

Lógicos
Not
And
Or

Ejemplo:

Imagine que quiere comprar dos productos, uno de 200 pesos y otro de 500 pesos, a los cuales habrá que aplicarles el 15 de impuestos. Vea cómo se resolvería la siguiente expresión compuesta.

200 + 500 * 1 + 0.15	Se resolverían primero las expresiones que involucran la multiplicación.
200 + 500 + 0.15	Luego se resolverán las sumas, en orden de izquierda a derecha.
700 + 0.15	
700.15	Expresión resuelta.

El ejemplo anterior demuestra cómo la precedecia de los operadores nos puede llevar a errores de cálculo difíciles de detectar. Lo que realmente queríamos era sumar el precio de los productos y multiplicarlos por 1.15, cosa que no sucedió.

Precedencia posicional

La *precedencia posicional* se presenta cuando se tienen varias expresiones que involucran operadores de misma precedencia implícita, y consisten en que se resolverán las expresiones de izquierda a derecha.

El ejemplo anterior ilustró la precedencia posicional, dado que al encontrarse puras sumas (operadores de igual precedencia) se resolvieron en orden de izquierda a derecha.

Precedencia explícita

La *precedencia e pl cita* es aquella que se provoca mediante el uso de paréntesis.

Aquello que se encierra en paréntesis en una expresión es obligado a resolverse sin respetar otras reglas de precedencia respecto a lo que está afuera de los paréntesis, sin embargo, lo que está entre paréntesis sigue sujeto a las reglas de precedencia implícitas y posicionales.

Veamos cómo podríamos resolver el ejemplo que ofreció resultados incorrectos.

Ejemplos:

Imagine que quiere comprar dos productos, uno de 200 pesos y otro de 500 pesos, a los cuales habrá que aplicarles el 15 de impuestos. Vea cómo se resolvería la siguiente expresión compuesta.

(200 + 500) * (1 + 0.15)	Se resolverían primero las expresiones que se encuentren entre paréntesis. Como son sumas, se resuelven atendiendo precedencia posicional.
(700) * (1 + 0.15)	
(700) * (1.15)	Finalmente queda la multiplicación que realmente pretendemos.
805	Expresión resuelta.

No olvide que por más larga que parezca una expresión, todos los operadores actúan sobre dos expresiones, y de dos en dos, hasta dejar un solo valor final.

Ejercicio 05.01

Aplicación de operadores y sus reglas de precedencia

Resuelva las expresiones, una por una, en orden de precedencia. Subraye en cada línea la expresión que ha de resolverse. Vea el ejemplo que se propone.

23 + 25 * 25 + 2 / 10

23 + 625 + 2 / 10

23 + 625 + 0.2

648 + 0.2

648.2

1. **48 + 25 * 10 / 4 + 5**

2. **50 / 5 > 30 ∨ 2 = 2**

3. **(20 + 30) / (10 * (4 / 2))**

4. **¬ (False ∨ True)**

5. ¬ (“a”=”A” ∨ (20 * 30 * 50 > 10 * 40) ∧ (30 ^ 2 > 15 ^ 4))

FIN DEL EJERCICIO ✱

Ejercicio 05.02

Representación de expresiones complejas operadores y elementos de lógica simbólica

Represente, utilizando los operadores y los símbolos de lógica simbólica, las expresiones que se proponen. Vea el ejemplo.

otal es igual a multiplicar ni a es por Precio ni a , y aplicarle al producto el 15 de impuestos.

Total ← (Unidades * Precio) * (1 + 0.15)

1. onto o ra es igual a PrecioPro cto multiplicado por el Precio nitario, menos el onto esc ento que se haya autorizado.

2. A tori aci n re ito sólo es posible si onto o ra es menor que r ito is onible.

3. ec a o re ito es lo contrario a A tori aci n r ito.

4. A tori aci nA to tica es posible si i ite r ito es "Ilimitado" o si istorial re iticio es "Perfecto", siempre y cuando i o ar eta no sea "Adicional".

5. Una computadora ( ) tiene un 98 de grado de disponibilidad, y la conexión a Internet de banda ancha que utiliza ( ) tiene un 87 de disponibilidad. La computadora y la conexión a Internet se utilizan de forma conjunta. Si el horario de labores de la persona que los utiliza es de 8 horas, ¿Cuánto tiempo, en horas, puede disponer realmente de la información publicada en internet (is onibili a n or aci n)?

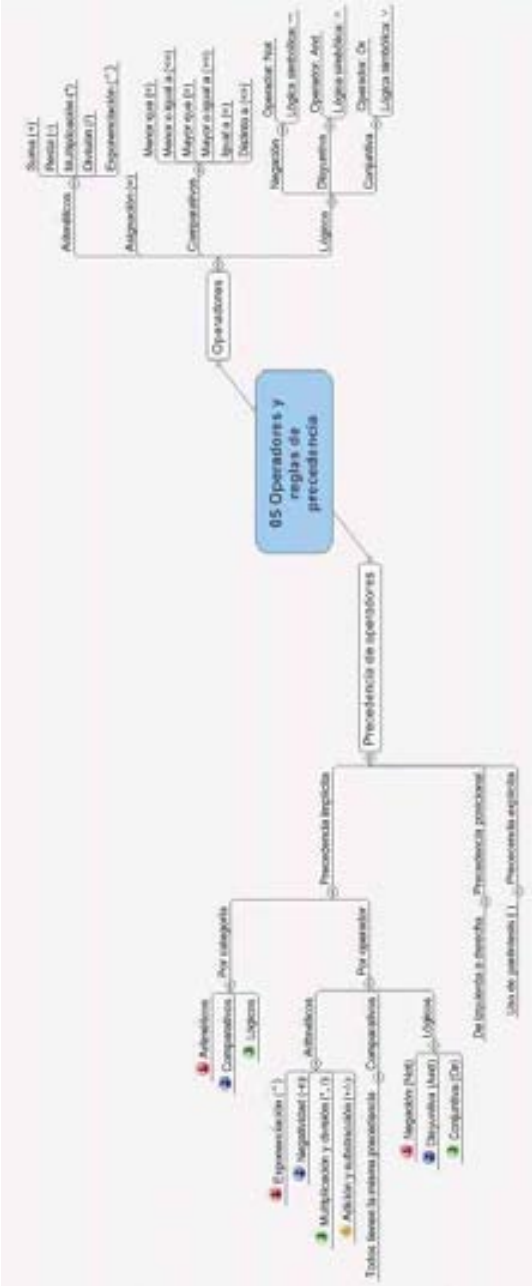
   La disponibilidad de un sistema de dispositivos (dos o más dispositivos que trabajan de forma conjunta) se calcula multiplicando los porcentajes de grado de disponibilidad de cada uno de ellos, y el resultado siempre será menor a la disponibilidad del menor.

   Para calcular el tiempo de disponibilidad de un servicio (en este caso el acceso a la información publicada en Internet), sólo basta con multiplicar el tiempo en que es requerido el servicio por la disponibilidad del sistema de dispositivos.

   Realice la representación para calcular
   is onibili a n or aci n.

FIN DEL EJERCICIO ✱

Mapa mental del capítulo



Terminología

Expresión lógica compuesta, 80
Expresión lógica simple, 80
Operador de asignación, 78
Operadores aritméticos, 76
Operadores comparativos, 78
Operadores lógicos, 79
Operadores, 76
Precedencia explícita, 82
Precedencia implícita por categoría, 81
Precedencia implícita por operador, 81
Precedencia implícita, 81
Precedencia posicional, 82
Precedencia, 80
Preferencia, 80

Preguntas

5.1 ¿Será posible automatizar proceso sin la utilización de operadores? Explique su respuesta.

5.2 En las reglas de precedencia implícita por categorías, se menciona que se resuelven primero las expresiones aritméticas que las comparativas ¿Qué problema habría si la precedencia fuera a la inversa?

5.3 Suponga que tiene la siguiente expresión.

$A > B \vee B>D \leftarrow X$

Complemente la tabla siguiente.

Si A	Si B	Si D	Entonces X
4	3	2	
3	4	2	
3	4	5	

Notas:

Examen rápido

Parte 1 : Marque co n una cruz el inciso que considere que responde a la pregunta.

1) Son los operadores que generan una nueva expresión numérica a partir de otras.
(a) Aritméticos
(b) Lógicos
(c) De asignación

2) Es el único operador que puede pertenecer a dos categorías.
(a)
(b)
(c)

3) Es el valor que indica ausencia de valor.
(a) 0
(b) Null
(c) Blan

4) Es la precedencia que aplica mediante el uso de paréntesis.
(a) Posicional
(b) Explícita
(c) Dominio

5) Aritméticamente, es el operador de mayor precedencia.
(a) Negatividad
(b) Multiplicación División
(c) Exponenciación

Parte 2: Coloque en la línea la letra «F» si la sentencia es falsa, y «V» si la sentencia es verdadera.

6) La precedencia sólo aplica para expresiones complejas.

_____ **7)** "A" < "a".

8) La precedencia posicional se asigna de izquierda a derecha.

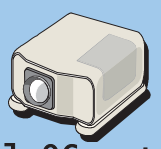
_____ **9)** La precedencia implícita por operador aplica para la siguiente expresión: 2*4 > 7.

_____ **10)** El operador Not sólo aplica para una sola expresión lógica.

_____

6

Algoritmos para el análisis de casos reales



lp06.ppt

Objetivos: Aprenderá a realizar análisis de casos reales de negocios y crear algoritmos.

1. Aprenderá el procedimiento de creación de programas de cómputo.
2. Aprenderá el proceso en que se implementa el ciclo de desarrollo en el mundo real.
3. Aprenderá a analizar casos del mundo real para determinar requerimientos de desarrollo de los clientes.
4. Aprenderá a estructurar algoritmos de forma documental.

www.detodoprogramacion.com

Contenido

Procedimiento de creación de un programa 95
Ciclo de desarrollo 95
Relevancia de las fases 97
Implementación práctica del ciclo de desarrollo 98
Exposición de casos prácticos 100
Caso 1: Mensaje 101
Caso 2: Niveles de servicio 101
Caso 3: Muestra de una tabla de multiplicar 102
Caso 4: Muestra las tablas de multiplicar del 1 al 5 102
Caso 5: Análisis de promedios 102
Analizar el caso real 102
Determinar de forma clara los requerimientos del cliente 103
Determinar el alcance del programa 106
Determinar la interfaz y el comportamiento de un sistema 108
Algoritmos 109
Analizando los casos y diseñando sus algoritmos 110
Caso 1: Mensaje 110
Caso 2: Niveles de servicio 111
Caso 3: Muestra de una tabla de multiplicar 114
Caso 4: Muestra las tablas de multiplicar del 1 al 5 116
Caso 5: Análisis de promedios 118
Comprobando habilidades en el análisis de casos y el diseño de algoritmos.... 120
Mapa mental del capítulo 130
Terminología 131
Preguntas 131
Examen rápido 133

na gran cosa siempre es una suma de pe ue as cosas.

A partir de este capítulo comenzaremos a ver las técnicas de programación de aplicaciones. En los capítulos anteriores sólo cubrimos los aspectos teóricos que nos permitirán entender las herramientas de desarrollo que estaremos utilizando.

Procedimiento de creación de un programa

La *función de programación* consiste en una serie de pasos que nos llevan desde que analizamos una situación de la vida real, hasta que vemos ejecutarse sin errores el programa que automatiza las tareas.

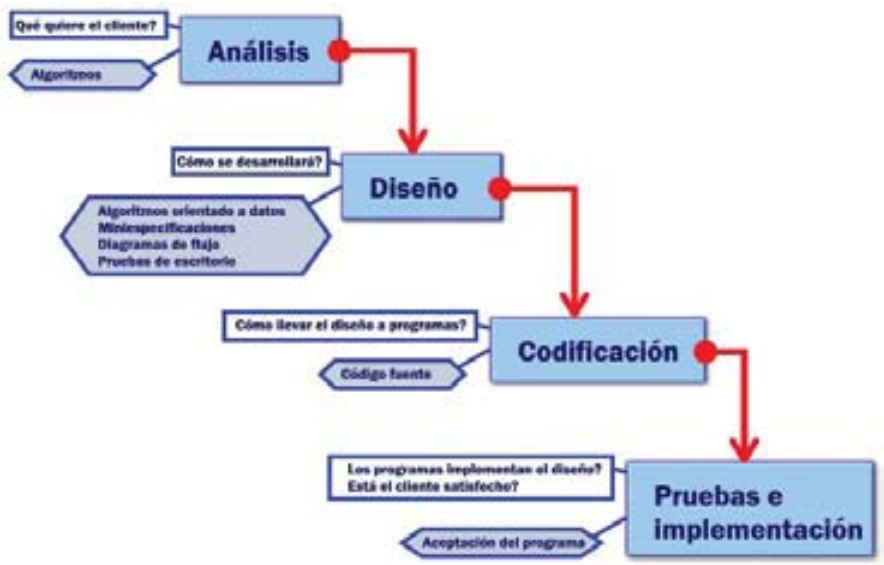
Ciclo de desarrollo

El *ciclo de desarrollo* de programas es muy simple, e involucra las fases siguientes: Análisis, Diseño, Codificación, y Pruebas e implementación. A grandes rasgos mencionaremos el objetivo primordial de la fase, así como sus entregables básicos, que son los documentos que resultan de haber desarrollado la fase en cuestión.

La representación más sencilla de este ciclo es la conocida como *modelo de cascada* (*aterfall model*), que luce como sigue.

Figura 06.01

Modelo de cascada del ciclo de desarrollo



- *ase de an lisis.*

  Es la fase de desarrollo en la cual se analiza una situación del mundo real, para entenderla a fondo, determinando los requerimientos del cliente.

  Responde a la pregunta: ¿Qué quiere automatizar el cliente?

  Entregables: Algoritmos.

- *ase de dise o.*

  Es la fase del desarrollo que traduce los requerimientos del cliente a una forma estandarizada y abstracta que sirva como materia prima para la codificación de aplicaciones.

  Responde a la pregunta: ¿Cómo se puede automatizar lo que quiere el cliente?

  Entregables: Algoritmo orientado a datos, miniespecificaciones, diagramas de flujo, pruebas de escritorio.

- *ase de codificación.*

  Es la fase del desarrollo que traduce los documentos de diseño a su representación en código fuente, utilizando un lenguaje de programación determinado.

  Responde a la pregunta: ¿Cómo se representan las especificaciones de diseño en un lenguaje de programación dado?

  Entregables: Código fuente del programa.

- *ase de pruebas e implementación.*

  Es la fase del desarrollo que corrobora si los programas cubren las especificaciones de diseño, y si el cliente está satisfecho con la forma en que el programa trabaja y con los resultados que proporciona.

  Responde a la pregunta: ¿Los programas cubren a la perfección las especificaciones de diseño?

  ¿Está el cliente satisfecho con la interfaz del programa y los resultados que ofrece?

  Entregables: Aceptación del programa.

NOTA

Existen otros modelos más actuales que proponen entregas parciales basadas en prototipos. Estos modelos, llamados modelos en espiral, proponen dedicar todas las fases del modelo de cascada atendiendo funcionalidades específicas, mostrar el resultado al cliente, obtener sus opiniones, hacer las modificaciones pertinentes, para continuar con otras funciones nuevas. El proceso se repite de manera cíclica, hasta concluir con todas las funciones de la aplicación. Estos modelos tienen la ventaja que se pueden detectar diferencias entre el producto y los requerimientos desde fases tempranas. Nosotros seguimos explicando el modelo en cascada porque a fin de cuentas, los modelos en espiral son uniones de modelos en cascada entregados de forma gradual.

Relevancia de las fases

Hay ciertas tareas que pueden realizarse manualmente, mientras que hay otras actividades que se tienen que realizar en la computadora, de forma necesaria. Paradójicamente, las actividades manuales son las más importantes en el proceso de automatización.

ada es tan in til como hacer eficientemente lo ue no debe hacerse.

Peter Druc er

Análisis. Las actividades relacionadas con el análisis son principalmente trabajo de campo y se realizan mejor de forma manual. Es un hecho que el analista de sistemas es el mejor pagado de todo el proceso de desarrollo; esto se debe a que el analista es el que determina qué se ha de diseñar, codificar y probar.

Es en la fase de análisis donde se encuentran las soluciones a los problemas de automatización, por lo que hay que ser muy analíticos y sistemáticos en determinar qué es lo que el cliente necesita, para no caer en excesos, pero sin omitir nada.

Si se comete un error en la fase de análisis, los requerimientos están equivocados; el diseñador de sistemas diseña sobre una situación que el cliente no necesita, el programador codifica sobre un diseño que está de inicio equivocado, las pruebas se hacen sobre un programa que eficientemente automatiza aquello que ni siquiera debería haberse automatizado, y al momento de validar la satisfacción del cliente, éste nos dice que eso no era lo que pidió. En resumen, se trabajó mucho sobre algo que no sirvió, y hay que iniciar todo el proceso nuevamente.

Diseño. Suponiendo que el análisis está bien realizado, el diseño es la segunda fase en importancia, dado que en el diseño se realiza el trabajo de abstracción de los resultados del análisis, y se pone en formato técnico entendible para los programadores.

En esta fase es importante evitar el sesgo y la falta de precisión. El trabajo de diseño sólo representa en forma técnica los requerimientos del cliente, pero no le agrega ni le quita nada a los requerimientos. Un vicio muy común es que los programadores que hacen diseño modifican los requerimientos del cliente con la finalidad de facilitar el proceso de codificación: esto es un error. De hecho, al momento de diseñar podemos no saber en qué lenguaje habremos de codificar, por lo cual esa práctica es innecesaria y puede provocar que lo obtenido no sea lo que el cliente solicitó.

Codificación. Si el diseño se realizó correctamente, el resultado del diseño puede proporcionarse a cualquier desarrollador para la elaboración de los programas, sin importar qué lenguaje de programación utilice. La elección del lenguaje, junto con la habilidad del programador para manejarlo, determinarán el tiempo de desarrollo.

La función de codificación se encarga de transformar a código fuente lo que expresan los documentos derivados del diseño, de forma fiel y sin sesgos. Su relevancia radica en que las fases más costosas y exhaustivas del desarrollo ya están hechas (análisis y diseño), pero pueden ser inútiles si el programador es incapaz de producir el programa que ya fue especificado por el diseño.

La función de programación es la peor pagada en todo el proceso de desarrollo. Esa situación es provocada principalmente porque existe mucha oferta de servicios de programación en el mercado, sin embargo, es un muy buen trabajo para las personas que inician en la profesión del desarrollo, ya que el programador tiene contacto con toda la documentación de análisis y diseño, y puede aprender el oficio de gente más experimentada. El programador puede aprender bastante, hasta que su experiencia le permita ser el analista o diseñador de soft are.

Pruebas e implementación. Esta fase es importante para el cliente, más que para el desarrollador. En esta fase el cliente comprueba si los requerimientos iniciales de información son cubiertos por el programa. De ser así, se realiza la implementación del programa, que no es otra cosa que instalarlo en ambientes de producción.

Su relevancia radica en que, si se encuentran discrepancias entre lo que el cliente deseaba y el programa obtenido, disparará de nuevo todo el proceso de desarrollo, a fin de realizar ajustes a lo analizado, diseñado o programado.

Implementación práctica del ciclo de desarrollo

Las fases de desarrollo descritas anteriormente sólo tienen valor académico. La realidad de las cosas es que cada una de las fases requiere de una serie de herramientas para lograr su cometido.

En este libro proponemos una forma de implementar el ciclo de desarrollo, a través del uso de sencillas herramientas. La mayoría de las cuales pueden realizarse de manera manual, dejando el uso de la computadora reservada a las labores de codificación. Por supuesto, algunas de las actividades que mencionamos como manuales pueden ser realizadas apoyándonos en soft are especializado que nos puede simplificar la vida.

A continuación se muestran las herramientas que sugerimos para implementar el ciclo de datos.

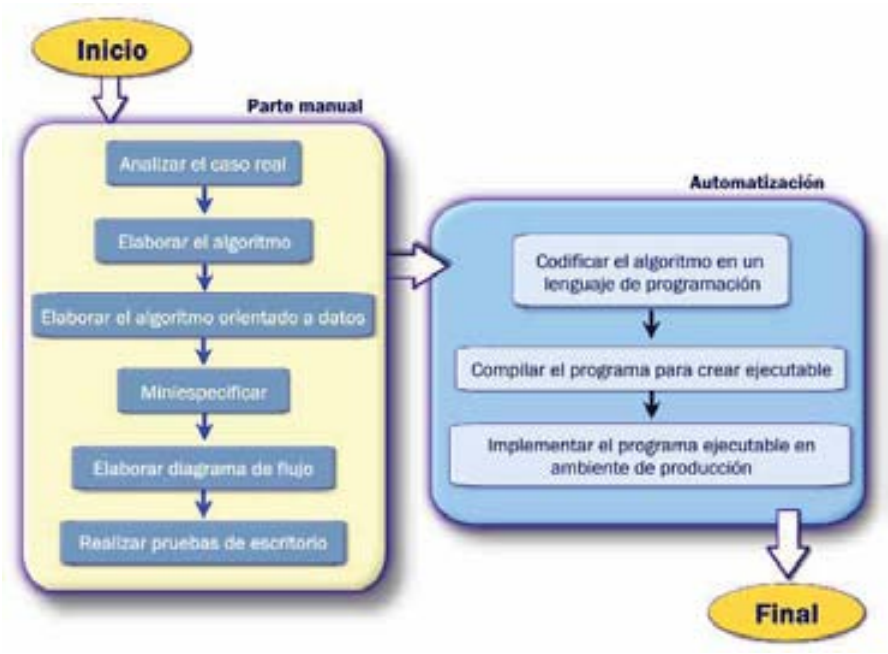
Fase	Actividad / herramienta	Implementación	Software sugerido (en su caso)
Análisis	Analizar el caso real	Manual	Mind Manager para los mapas mentales. Microsoft Visio para la diagramación. Procesador de texto, para registrar la información que se recopila.
	Algoritmo	Manual	Cualquier procesador de textos.

Fase	Actividad / herramienta	Implementación	Software sugerido (en su caso)
Diseño	Algoritmo orientado a datos	Manual	Cualquier procesador de textos. Uso de los símbolos establecidos en la lógica simbólica, y estándares para la especificación de dominios.
	Miniespecificación	Manual	Cualquier procesador de textos.
	Diagrama de flujo	Manual / Asistida	Microsoft Visio, para la diagramación compleja. Raptor, para la diagramación.
	Pruebas de escritorio	Manual / Asistida	Raptor, para la ejecución de pruebas de escritorio y monitoreo de los valores que toman las variables.
Codificación	Codificación de algoritmos	Asistida	Editores de texto plano. Herramientas de desarrollo asistido y entornos integrados de desarrollo, como Microsoft Visual Studio y como DreamWeaver.
	Compilación	Asistida	Herramientas de desarrollo (SDK). Herramientas de desarrollo asistido y entornos integrados de desarrollo, como Microsoft Visual Studio y como DreamWeaver.
Pruebas e implementación	Pruebas e implementación	Manual	Reuniones con el cliente para la revisión y la capacitación en el uso del programa.

Gráficamente podríamos representar al proceso de la siguiente manera.

Figura 06.02

Modelo del proceso de implementación del ciclo de desarrollo



a que conoce el proceso de desarrollo, es necesario tener algunos requerimientos sencillos que nos lleven a probar nuestras habilidades de desarrollo.

Exposición de casos prácticos

Si uno observa los libros de lógica de programación, o bien los de programación elemental, es claro que se esmeran en colocar el mayor número posible de ejercicios, como si ello dejara mayor conocimiento. El resultado ha sido nefasto en muchas ocasiones, dado que el lector de los libros tiene dos problemas al mismo tiempo: aprender las técnicas de programación que son cosa nueva para ella o él, y además tener que estar lidiando con los múltiples ejercicios.

En este libro adoptamos otra posición. En todo el texto sólo se cubrirán solamente 5 programas elementales. Todas las herramientas del proceso serán comprobadas utilizando ese reducido número de ejercicios.

Lo que esperamos es que el lector se familiarice profundamente con las situaciones que se exponen como caso, que entienda de qué se tratan, y que no se vuelva a preocupar por entender el problema. Lo que queremos es que se preocupe por dominar la técnica, dado que la vida le proporcionará una cantidad aplastante de problemas que solucionar.

A fin de cuentas, aprender muchos ejemplos tales como "calcule el área de un triángulo" o "adivine un número aleatorio" no genera experiencia útil en el plano laboral; no me han solicitado un programa parecido en mis muchos años de experiencia como programador, dado que ninguna organización se dedica a calcular áreas de triángulos o adivinar números aleatorios. Sin embargo, aprender la técnica de programación sí genera experiencia útil en el plano laboral. Eso es cosa de todos los días.

Los casos prácticos que proponemos son los siguientes. Si logra dominar la técnica de programación con ellos, podrá enfrentar la mayoría de los programas en la vida real.

Caso 1: Mensaje

Este programa tiene como finalidad comprobar técnicas y herramientas de desarrollo. Lo que el programa debe hacer es preguntar por un mensaje de saludo, para posteriormente mostrar el mensaje que se le proporcionó.

Utilidad del programa:

1. Comprueba la estructura básica de los programas en cualquier lenguaje.
2. Comprueba los estatutos de asignación de datos.
3. Comprueba los estatutos de recopilación de datos.
4. Comprueba los estatutos para el desplegado de datos existentes.

Caso 2: Niveles de servicio

Se desea saber la disponibilidad del servicio de Internet para una persona, expresado en horas de disponibilidad.

Dicho nivel de servicio depende del grado de disponibilidad que ofrece la computadora que utiliza, así como del grado de disponibilidad del enlace a Internet que utiliza la máquina.

El horario de labores de la persona que utiliza la computadora y el enlace a Internet es de 8 horas.

Para poder realizar los cálculos adecuadamente, tome en cuenta que la disponibilidad de un sistema de dispositivos (dos o más dispositivos que trabajan de forma conjunta) se calcula multiplicando los porcentajes de grado de disponibilidad de cada uno de ellos, y el resultado siempre será menor a la disponibilidad del menor.

Para calcular el tiempo de disponibilidad de un servicio (en este caso el acceso a la información publicada en Internet), sólo basta con multiplicar el tiempo en que es requerido el servicio por la disponibilidad del sistema de dispositivos.

Utilidad del programa:

1. Comprueba la realización de cálculos para la generación de datos nuevos.

Caso 3: Muestra de una tabla de multiplicar

El programa debe mostrar cualquier tabla de multiplicar que le solicitemos. La única restricción es que la tabla deberá ser de un número mayor o igual a 1, y menor o igual a 10.

Si se pide una tabla de un número que no esté entre el 1 y el 10, el programa debe indicar, mediante un mensaje, que no puede continuar.

Utilidad del programa:

1. Comprueba el uso de estatutos condicionales y estatutos de ciclo.
2. Comprueba el uso de los contadores.

Caso 4: Muestra las tablas de multiplicar del 1 al 5

El programa debe mostrar todas las tablas de multiplicar del 1 al 5.

Utilidad del programa:

1. Comprueba el uso de estructuras anidadas.

Caso 5: Análisis de promedios

El programa debe recopilar 5 números, para después mostrarlos indicando si están o no por encima de su promedio. Al final deberá enviarse un resumen en donde se indique la suma total de los números, y el promedio calculado.

Para la recopilación de datos utilice un arreglo.

Utilidad del programa:

1. Comprueba el uso de acumuladores.
2. Comprueba el uso de los arreglos.

Analizar el caso real

El *an lisis* consiste en descomponer un todo en sus partes, con el fin de comprender la forma en que el todo se constituye y comporta.

Dentro del contexto del desarrollo de aplicaciones y programas, el análisis tiene como finalidad lo siguiente.

1. Determinar de forma clara los requerimientos del cliente.
2. Determinar el alcance del programa.
3. Determinar la interfaz y comportamiento esperado del programa.

Determinar de forma clara los requerimientos del cliente

Determinar los *re uerimientos del cliente* consiste en llegar a conocer y documentar lo que el cliente espera de nuestra aplicación. Estos son sólo algunos ejemplos de lo que el cliente puede estar persiguiendo con un programa.

1. Automatizar un proceso manual.
2. Mejorar un proceso automatizado existente.
3. Aumentar rapidez.
4. Reducir costos.
. Incrementar los ingresos.
6. Habilitar un servicio basado en información.
. Documentar transacciones.
. Actualizar bases de datos.

Importancia del a nálisis. Para ejemplificar más la importancia del análisis, imagine que acude a un restaurante de comida rápida. Llega con ganas de una deliciosa hamburguesa con papas, queso y un refresco bien frío. El mesero, después de unos minutos, lleva a su mesa el más delicioso helado disponible en el restaurante.

Estamos de acuerdo que, aunque el helado es muy bueno, no fue lo que pidió y por tanto el restaurante ha fracasado en su intento de satisfacer su necesidad inicial de la forma más rápida posible. Aquí surgen varias posibilidades: si acepta el helado se quedará con las ganas de la hamburguesa; si no lo acepta, no lo pagará, y el restaurante perderá dinero. Si además usted sigue deseando la hamburguesa y está dispuesto a esperar más, tendrá que invertir más tiempo del que tenía previsto; aunque la hamburguesa esté muy buena se sentirá decepcionado del lugar.

Lo mismo puede pasar con las aplicaciones de cómputo. Uno debe estar muy seguro de lo que el cliente pide, a fin de poder entregarle lo que pidió.

Es indispensable conocer qué espera el cliente, dado que el éxito del programa radica en satisfacer precisamente la necesidad que él o ella valora. De nada servirá que nuestro programa sea maravilloso haciendo aquello que el cliente no esperaba del programa, ya que no le asignará valor. Generalmente los clientes no pagarán por ello y usted perderá su tiempo y su dinero.

Diferenciando al c liente y al c liente. Uno de los errores más comunes al analizar es tomar los requerimientos de la persona inadecuada.

De acuerdo a (*nformation echnology nfraestructure ibrary*), el término *cliente* aplica a la persona que recibe los beneficios finales del uso de la tecnología, y generalmente es el que paga por ello. El *cliente*, en cambio, es el que utiliza la tecnología en su trabajo diario. Generalmente el cliente sabe hacia dónde se dirige la organización y piensa en términos de negocio, mientras que el cliente muchas veces sólo ve la actividad que desarrolla para la organización, y busca su comodidad.

Una forma sencilla de descubrirlos es planteando la siguiente pregunta ¿Justificaría la compra de monitores de plasma para todas las computadoras antiguas de la organización? Probablemente el cliente le diga que estaría excelente, pero el cliente, que tendría que pagarlos, preguntará cuál es el beneficio de invertir en ello.

Una buena práctica es asegurarse que está recopilando los deseos tanto del cliente como del cliente. Si los requerimientos que trasmite el cliente son opuestos a los requerimientos del cliente, haga caso al cliente, que será quien pague por el programa. En caso de que las diferencias sean mínimas, trate de conciliar los requerimientos.

NOTA

ITIL es una biblioteca de mejores prácticas para la prestación de servicios de tecnología de la información, que garantiza que las inversiones en TI realmente tengan impacto en los servicios y procesos de las organizaciones. Es internacionalmente reconocida y rápidamente está reuniendo adeptos en las empresas más importantes del mundo.

Preparación previa. Los analistas de sistemas siempre enfrentarán el problema de tener que automatizar algo que desconocen. De inicio el cliente trata de comunicarse con los analistas de forma clara, pero es cuestión de tiempo para que comience a decir cosas que no nos quedan claras.

Partimos del supuesto de que un programa de cómputo nunca existe por sí mismo.

Siempre busca automatizar cosas relacionadas con otras disciplinas del conocimiento distintas a la informática, y es necesario que como analista tenga un conocimiento general de lo que necesita automatizar.

Esa necesidad hace de los analistas programadores profesionales unos verdaderos todólogos, ya que terminan conociendo bastante bien aquello que automatizan; no es extraño toparnos con un analista que maneja bastante bien los términos médicos, comerciales y de manufactura, porque ha tenido que automatizar procesos de dichas disciplinas.

El proceso de adquirir conocimiento de una disciplina consiste en:

1. Identificar la disciplina en la que se ubica el problema.
2. Conocer la terminología especializada (conocido como *argot*).
3. Conocer las teorías y principios sobre las que descansa la disciplina.
4. Conocer los procesos inherentes a la disciplina.

Suponga que le piden automatizar un control de inventarios.

Primero tendríamos que determinar la disciplina en la cual necesitamos documentarnos. En nuestro caso la disciplina es la administración de inventarios.

Disciplina	A inistraci n e in entarios rea de la administración que se encarga de la planeación, organización, dirección y control de las actividades tendientes al asiento de los bienes y demás cosas pertenecientes a una persona o comunidad, hecho con orden y precisión.

Podemos conseguir un libro o documento relacionado con el tema, y nos encontraremos que para desarrollar el programa planteado, es necesario saber:

1. Qué son los bienes.
2. Qué controles se tienen sobre los bienes.
3. Que son las entradas de bienes.
4. Qué son las salidas de bienes.
. Qué son los movimientos de bienes.
6. Qué es el punto de reorden.
. Qué debemos entender por existencia, etcétera.

Recomendamos ampliamente que estudie a fondo la terminología especializada, en primera instancia, para después aventurarse en el entendimiento de los procedimientos.

a que conocemos los términos relacionados, podemos ver qué teorías de la disciplina nos pueden ser útiles en el manejo de dichos conceptos.

Proceso de análisis. El proceso de analizar cosas es una actividad que como seres racionales, somos capaces de hacer prácticamente desde que nacemos. Cuando algo es tan grande como para comprenderlo rápidamente, tendemos a ver sus partes para irlas reconociendo una por una, asociándolas a cosas que ya conocemos, hasta ir dando significado al todo.

Para realizar el análisis de un caso real, lo primero que debe hacer es despojarse de todo rastro de orgullo que le impida preguntar lo que no sabe.

Sugerimos el siguiente procedimiento para analizar algo.

1. Investigue cuál es el área del conocimiento al que pertenece el problema que se desea automatizar.
2. Documéntese en el área del conocimiento al que pertenece el problema que se desea automatizar, antes de acudir con el cliente a resolver dudas.
3. Vea el todo y trate de entenderlo.
4. Separe las cosas que no entiende de las que sí entiende.
. Enumere las cosas que no entienda, y pregunte al que sabe.

6. Comience por preguntar aquellas cosas que le sugieren alguna idea vaga o inexacta, y termine por aquello que no tiene ni la más remota idea de qué es.

. No olvide preguntar cuál es la relación de las cosas entre sí.

Validación de requerimientos. Una actividad indispensable en las tareas de análisis es preguntar al cliente si los requerimientos que hemos recopilado son los correctos. No debemos asumir que entendimos bien: hay que comprobarlo.

Una forma sencilla de hacerlo es elaborando un informe que detalle los requerimientos recopilados, programar una reunión con el cliente y exponerle nuestro trabajo, a fin de que lo apruebe.

Una lista de unidades de funcionalidad puede servir para ello. Más adelante encontrará en qué consiste.

Determinar el alcance del programa

El *alcance de un programa* consiste en la cantidad de funcionalidad mínima, suficiente y necesaria para que el cliente considere a la aplicación como un producto terminado.

Es necesario delimitar claramente los requerimientos del cliente, y traducirlos en *unidades de funcionalidad* (llamados también *puntos de función*, o *function points*).

Las unidades de funcionalidad son aquellos resultados y comportamientos específicos que el cliente desea que el programa cubra. La aplicación está concluida si todas las unidades de funcionalidad están cubiertas. De hecho, la fase de pruebas consiste en verificar que todas las unidades de funcionalidad estén presentes.

Cada unidad de funcionalidad deberá tener un identificador, una prioridad y una obligatoriedad. En caso de que el desarrollo de la aplicación tenga problemas de presupuesto o de tiempo, será posible eliminar funcionalidad que no es tan necesaria.

NOTA

Los puntos de función son un concepto propio de la ingeniería de software, y no existe una regla clara que determine el alcance de un punto de función. Hemos notado que un punto de función es algo que el cliente identifica como actividad que es posible desarrollar con la aplicación. Esa actividad es independiente y autónoma, respecto a las demás, aunque forme parte de un todo .

Volviendo al ejemplo del control de inventarios, se podrían generar las siguientes unidades de funcionalidad.

ID	Unidad de funcionalidad	Prioridad	Obligatoriedad
1	Registrar bienes	Alta	Si
2	Registrar movimientos de entrada	Alta	Si
3	Registrar movimientos de salida	Alta	Si
4	Emitir listado de bienes y existencias	Alta	Si
5	Calcular automáticamente el punto de reorden de cada bien	Media	No
6	Informar entradas y salidas mensuales	Alta	Si
7	Informar de manera ejecutiva el costo de almacenaje	Baja	No

Al finalizar, el cliente esperará que la aplicación cuente con toda la funcionalidad anteriormente descrita. Vea cómo las unidades de funcionalidad inician con un verbo en infinitivo, lo que indica una acción posible a realizar a través del programa.

Cada unidad de funcionalidad representa para el cliente una actividad posible utilizando el programa.

La especificación de prioridad y obligatoriedad nos permiten determinar los elementos de un desarrollo completo o incompleto. Si falta alguna unidad de funcionalidad obligatoria, el programa no puede considerarse concluido.

Si el presupuesto o el tiempo es recortado, las especificaciones de prioridad y obligatoriedad nos permiten ir liberando carga de trabajo. Primero se eliminarán del plan de trabajo las unidades de funcionalidad de prioridad baja y no obligatorias, después las de prioridad media no obligatorias, después las de prioridad alta no obligatorias. Si tenemos que eliminar hasta las unidades obligatorias, hay que reconsiderar dar por cancelado el proceso de desarrollo. Si eso no es una opción, deberemos ir quitando en base a prioridad (baja, media y alta) a las unidades obligatorias.

Como analista, no deberá de incluir ni más ni menos funciones que las acordadas; eso le permitirá calcular el tiempo de desarrollo y el costo del mismo. Sólo después de haber delimitado el alcance se podrá dar un estimado de tiempo y costo.

NOTA

Una mala costumbre de las organizaciones es que el cliente mismo determina cuánto quiere que cueste el desarrollo, y cuánto quiere que se tarde. Los trabajos de desarrollo siguen su curso, y han de tardarse y han de costar lo justo. El ideal es que un profesional de sistemas sea el que haga la estimación de tiempo y costo de desarrollar una aplicación, después de haber hecho un análisis de las unidades de función que el cliente requiere. Podemos identificar al desarrollador novato porque ofrece fechas de conclusión y costos de desarrollo sin haber hecho antes el análisis de requerimientos, lo cual conduce generalmente en proyectos desfasados en tiempo y presupuesto.

Determinar la interfaz y el comportamiento de un sistema

Es necesario que el cliente tenga al menos una idea de la forma en que el programa ha de lucir, así como la forma en que ha de comportarse.

Interfaz. La *interfaz* es el medio a través del cual el usuario y la aplicación se comunican. Es importante que al cliente se le proponga una distribución de interfaces, y que además se validen con los usuarios, que son los que finalmente utilizarán las interfaces.

El *objetivo de una interfaz* es proveer datos completos y correctos para el procesamiento y el almacenamiento. Una interfaz debe ser.

1. Sencilla de entender.
2. Fomentar su uso intuitivo.
3. Apegarse a los estándares del ambiente.
4. Validar los dominios.
. Tener una secuencia lógica de sus elementos.
6. Preguntar la cantidad mínima, suficiente y necesaria para el procesamiento o almacenamiento.

Figura 06.03

Ejemplo de una interfaz. Debe proveer toda la información necesaria para imprimir un documento.

Afortunadamente, muchas herramientas actuales de desarrollo permiten elaborar de forma gráfica interfaces, por lo cual mostrarle al cliente y al usuario las propuestas de interfaces es muy sencillo.

Simulación. Una herramienta muy útil en esta labor es la de elaborar una *simulación*, es decir, una representación breve del comportamiento que esperamos del programa; la simulación puede hacerse de forma documental.

Lo ideal en una simulación es mostrar la forma en que responde el programa ante unos determinados datos de entrada. Esto permite que el cliente pueda detectar cosas incorrectas y retroalimentar respecto a errores evidentes. Para la simulación es muy importante que el cliente proporcione un ejemplo de los resultados correctos que el programa deberá reproducir considerando unos datos de entrada determinados.

Si el tiempo lo permite, la simulación se puede realizar de forma automatizada a través de una aplicación no definitiva que muestre el *loo and feel* del trabajo que esperamos realizar, y los resultados que esperamos. A esta versión no definitiva se le llama *prototipo*.

Algoritmos

De acuerdo con la Real Academia Española, podemos definir a un *algoritmo* como un conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.

En términos de programación, un *algoritmo de programación* sería un conjunto ordenado y finito de asignaciones, procesos, cálculos y decisiones que permiten a un programa satisfacer una unidad de funcionalidad dada.

Si tuviéramos que realizar un programa que calculara el área de un cuadrado y la desplegara, tendríamos que el algoritmo sería el siguiente.

1. Inicio.
2. Preguntar la medida de un lado.
3. Calcular el área del cuadrado.
4. Desplegar el valor del área.
5. Fin.

Los algoritmos sirven para validar con el cliente la secuencia de operaciones que nos permitirán cubrir las unidades de función. Como son proporcionadas al cliente para su validación, no deben contener términos técnicos.

En un algoritmo el cliente puede hacernos precisiones que nos lleven a cambiar el orden de las operaciones a realizar, agregar o quitar operaciones, e incluso redimensionar el alcance de las unidades de función.

Analizando los casos y diseñando sus algoritmos

A partir de este momento, en cada capítulo se aplicarán las técnicas aprendidas con los 5 casos propuestos. Al final se incluirá un ejercicio que contiene 3 ejercicios con los cuales se podrá comprobar lo aprendido.

NOTA

Por ser programas muy elementales, la relación de unidades de función incluye una sola unidad. Generalmente se tienen varias unidades de funcionalidad cuando se trata de una aplicación compuesta por varios programas que al final se integran en uno solo. Se incluye la tabla como referencia de redacción.

Caso 1: Mensaje

Este programa tiene como finalidad comprobar técnicas y herramientas de desarrollo. Lo que el programa debe hacer es preguntar por un mensaje de saludo, para posteriormente mostrar el mensaje que se le proporcionó.

Análisis del caso

rea el conoci iento Ninguna.

An lisis e los ele entos el to o

1. Se debe preguntar un mensaje.
2. Se debe mostrar el mensaje.

Unidades de funcionalidad

ID	Unidad de funcionalidad	Prioridad	Obligatoriedad
1	Preguntar un mensaje, para luego desplegarlo.	Alta	Si

Interfaz

Escribe un mensaje a mostrar:

Aquí se muestra el mensaje

Algoritmo

1. Inicio.
2. Se pregunta un mensaje.
3. Se muestra el mensaje.
4. Fin.

Simulación

El programa se comportaría de la siguiente manera si le proporcionamos el mensaje “Aprenda”.

Escribe un mensaje a mostrar: Aprenda

Aprenda

Caso 2: Niveles de servicio

Se desea saber la disponibilidad del servicio de Internet para una persona, expresado en horas de disponibilidad.

Dicho nivel de servicio depende del grado de disponibilidad que ofrece la computadora que utiliza, así como del grado de disponibilidad del enlace a Internet que utiliza la máquina.

El horario de labores de la persona que utiliza la computadora y el enlace a Internet es de 8 horas.

Para poder realizar los cálculos adecuadamente, tome en cuenta que la disponibilidad de un sistema de dispositivos (dos o más dispositivos que trabajan de forma conjunta) se calcula multiplicando los porcentajes de grado de disponibilidad de cada uno de ellos, y el resultado siempre será menor a la disponibilidad del menor.

Para calcular el tiempo de disponibilidad de un servicio (en este caso el acceso a la información publicada en Internet), sólo basta con multiplicar el tiempo en que es requerido el servicio por la disponibilidad del sistema de dispositivos.

Análisis del caso

rea el conoci iento Cálculo de niveles de servicio.

An lisis e los ele entos el to o

1. Se desea conocer la disponibilidad del servicio de Internet, expresado en horas.
2. El grado de disponibilidad de la computadora influye en el grado de disponibilidad del servicio.
3. El grado de disponibilidad del servicio de enlace a Internet influye en el grado de disponibilidad del servicio.
4. El tiempo total en el que se requiere el servicio es de 8 horas.
. La disponibilidad de un sistema de dispositivos se calcula multiplicando los porcentajes del grado de disponibilidad de los mismos.
6. La computadora y el servicio de enlace a Internet trabajan como un sistema de dispositivos.

Unidades de funcionalidad

ID	Unidad de funcionalidad	Prioridad	Obligatoriedad
1	Calcular el tiempo de disponibilidad del servicio, en horas, y mostrar el resultado.	Alta	Si

Interfaz

Grado de disponibilidad de la computadora:

Grado de disponibilidad del enlace:

Tiempo laborable en horas:

La disponibilidad del servicio de Internet es de resultado horas.

Algoritmo

1. Inicio.
2. Se pregunta la disponibilidad de la computadora.
3. Se pregunta la disponibilidad del enlace.
4. Se pregunta el tiempo laborable en horas.
5. Se calcula la disponibilidad del sistema de dispositivos.
6. Se calcula el tiempo de disponibilidad del servicio.
7. Se muestra el tiempo de disponibilidad del servicio.
8. Fin.

Simulación

Supongamos que la computadora que la persona utiliza brinda una disponibilidad del 98 , es decir, el 98 del tiempo funciona. Por otro lado, se ha comprobado que el enlace a Internet de banda ancha que la persona utiliza brinda una disponibilidad del 87 , es decir, el 87 del tiempo se tiene comunicación.

Disponibilidad del sistema =

Disponibilidad de la computadora * Disponibilidad del enlace

Aquí un ejemplo de cómo se realizarían los cálculos.

DS = DC * DE

DS = 0.98 * 0.87

DS = 0.85

La disponibilidad del sistema sería del 85.26 . La disponibilidad en horas del servicio sería el 85.26 del tiempo laborable.

Disponibilidad en horas = Disponibilidad del sistema * Tiempo laborable

DH = DS * TL

DH = 0.8526 * 8

DH = 6.82

La respuesta del programa sería la siguiente:

```
Grado de disponibilidad de la computadora: 0.98
Grado de disponibilidad del enlace: 0.87
Tiempo laborable en horas: 8
La disponibilidad del servicio de Internet es de 6.82 horas.
```

Caso 3: Muestra de una tabla de multiplicar

El programa debe mostrar cualquier tabla de multiplicar que le solicitemos. La única restricción es que la tabla deberá ser de un número mayor o igual a 1, y menor o igual a 10.

Si se pide una tabla de un número que no esté entre el 1 y el 10, el programa debe indicar, mediante un mensaje, que no puede continuar.

Análisis del caso

rea el conoci iento Matemática elemental.

An lisis e los ele entos el to o

1. Se debe mostrar una tabla de multiplicar.
2. La tabla a mostrar debe ser solicitada.
3. La tabla solicitada debe estar entre el 1 y el 10.
4. Cualquier solicitud de tabla fuera del rango de 1 a 10 debe ser rechazada, provocando un mensaje de error.

Unidades de funcionalidad

ID	Unidad de funcionalidad	Prioridad	Obligatoriedad
1	Solicitar una tabla de multiplicar y desplegarla.	Alta	Si

Interfaz

Qué tabla deseas:

Aquí se muestra el resultado

Algoritmo

1. Inicio.
2. Se pregunta qué tabla se desea
3. Si la tabla está entre 1 y 10, entonces se imprime la tabla solicitada
4. Si la tabla no está entre el 1 y 10, entonces se muestra un mensaje de error.
5. Fin.

Simulación

Supongamos que nosotros le pedimos la tabla del 5, el programa debería mostrar lo siguiente.

Qué tabla deseas: 5

5 x 1 = 5

5 x 2 = 10

5 x 3 = 15

5 x 4 = 20

5 x 5 = 25

5 x 6 = 30

5 x 7 = 35

5 x 8 = 40

5 x 9 = 45

5 x 10 = 50

Supongamos que nosotros le pedimos la tabla del 25, el programa debería mostrar lo siguiente.

```
Qué tabla deseas: 25
Válido sólo del 1 al 10
```

Caso 4: Muestra las tablas de multiplicar del 1 al 5

El programa debe mostrar todas las tablas de multiplicar del 1 al 5, incluyendo un encabezado por tabla.

Análisis del caso

rea el conoci iento Matemática elemental.

An lisis e los ele entos el to o

1. Se deben mostrar las tablas de multiplicar del 1, 2, 3, 4 y 5.

Unidades de funcionalidad

ID	Unidad de funcionalidad	Prioridad	Obligatoriedad
1	Desplegar las tablas de multiplicar del 1 al 5.	Alta	Si

Interfaz

```
Muestra las tablas del 1 al 5 aquí
```

Algoritmo

1. Inicio.
2. Muestra el encabezado de la tabla del 1.
3. Se calcula la tabla del 1 y se muestra.
4. Muestra el encabezado de la tabla del 2.
5. Se calcula la tabla del 2 y se muestra.
6. Muestra el encabezado de la tabla del 3.
7. Se calcula la tabla del 3 y se muestra.

8. Muestra el encabezado de la tabla del 4.
9. Se calcula la tabla del 4 y se muestra.
10. Muestra el encabezado de la tabla del 5.
11. Se calcula la tabla del 5 y se muestra.
12. Fin.

Simulación

No se proporcionan datos. El programa es de sólo salida, y muestra lo siguiente.

```
TABLA: 1
1 X 1 = 1
1 X 2 = 2
1 X 3 = 3
1 X 4 = 4
1 X 5 = 5
1 X 6 = 6
1 X 7 = 7
1 X 8 = 8
1 X 9 = 9
1 X 10 = 10
TABLA: 2
2 X 1 = 2
2 X 2 = 4
...
...
...
5 X 6 = 30
5 X 7 = 35
5 X 8 = 40
5 X 9 = 45
5 X 10 = 50
```

Caso 5: Análisis de promedios

El programa debe recopilar 5 números, para después mostrarlos indicando si están o no por encima de su promedio. Al final deberá enviarse un resumen en donde se indique la suma total de los números, y el promedio calculado.

Para la recopilación de datos utilice un arreglo.

Análisis del caso

rea el conoci iento Matemática elemental.

An lisis e los ele entos el to o

1. El programa recopila 5 números.
2. El programa suma los números proporcionados.
3. El programa calcula el promedio de los números proporcionados.
4. El programa determina si cada uno de los números está por encima o no de su promedio.
. El programa muestra la suma total calculada.
6. El programa muestra la comparación de cada uno de los números respecto al promedio calculado.

Unidades de funcionalidad

ID	Unidad de funcionalidad	Prioridad	Obligatoriedad
1	Recopilar una serie de números, calcular el promedio de la serie, y mostrar cada uno de los números proporcionados, indicando si están o no por encima del promedio.	Alta	Si

Interfaz

Valor para el elemento 1: _____

Valor para el elemento 2: _____

Valor para el elemento 3: _____

Valor para el elemento 4: _____

Valor para el elemento 5: _____

<Resultado del programa>

Algoritmo

1. Inicio.
2. Se pregunta el valor del elemento 1.
3. Se pregunta el valor del elemento 2.
4. Se pregunta el valor del elemento 3.
5. Se pregunta el valor del elemento 4.
6. Se pregunta el valor del elemento 5.
7. Se realiza la suma de todos los elementos.
8. Se calcula el promedio de la serie de números.
9. Se muestra la lista de valores, informando si el número se encuentra por encima o no del promedio.
10. Se muestra el resultado de la suma.
11. Se muestra el resultado del promedio.
12. Fin.

Simulación

Supongamos que nosotros le proporcionamos los siguientes valores al programa: 10, 15, 25, 9, 4. La suma de los valores da 63. Dividiendo la suma entre 5 (número de elementos), se tiene un promedio de 12.6.

```
Valor para el elemento 1: 10
Valor para el elemento 2: 15
Valor para el elemento 3: 25
Valor para el elemento 4: 9
Valor para el elemento 5: 4

Número 10 menor al promedio
Número 15 mayor o igual al promedio
Número 25 mayor o igual al promedio
Número 9 menor al promedio
Número 4 menor al promedio
Suma: 63
Promedio: 12.6000
```

Ejercicio 06.01

Comprobando habilidades en el análisis de casos y el diseño de algoritmos

Elabore lo siguiente para cada uno de los casos que se propongan.

- ❒ Análisis del caso.
- ❒ Relación de unidades de funcionalidad.
- ❒ Interfaz.
- ❒ Algoritmo.

1.- Caso 6: Regla de tres simple

Elabore un programa que realice el cálculo de una regla de tres simple. Como sabe, dicha regla permite calcular la proporción de un número respecto a otro que actúa como referencia de la proporción. Este programa contesta preguntas del tipo ¿qué porcentaje de 200 es 135? El 200 sería el número base de la proporción, mientras que el 135 sería la proporción. La solución se calcula multiplicando la proporción por 100, y dividiéndolo entre el número base de la proporción.

Análisis del caso

rea el conoci iento

An lisis e los ele entos el to o

Unidades de funcionalidad

ID	Unidad de funcionalidad	Prioridad	Obligatoriedad
1			

Interfaz

Algoritmo

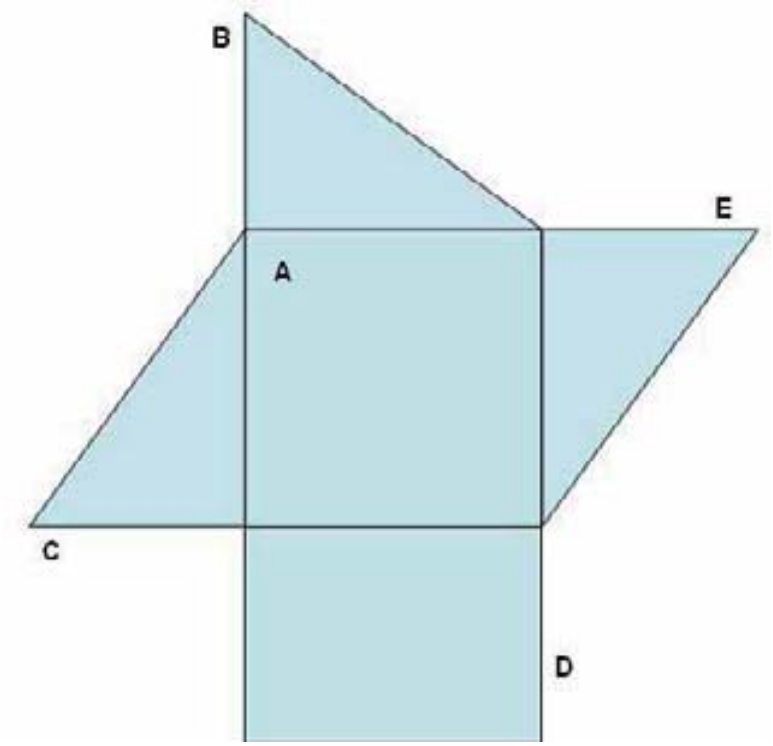
Simulación

Anote aquí el comportamiento que espera de su programa.

Anote aquí la interfaz de salida.

2.- Caso 7: Cálculo del área de un polígono compuesto

Observe la siguiente figura.



El programa debe calcular el área del polígono. La figura se compone de un cuadrado perfecto (A), tres triángulos rectángulos iguales (B, C, E), cuya base mide lo mismo que uno de los lados del cuadrado, y un rectángulo cuyo ancho mide lo mismo que uno de los lados del cuadrado, mientras que la altura mide lo mismo que la altura de los triángulos. Se pide que haga un programa que calcule el área total.

Por cultura general, sabemos que el área de un rectángulo se calcula multiplicando su base por su altura; el área de un triángulo rectángulo se calcula multiplicando su base por su altura, y luego dividiendo el resultado entre dos; finalmente, el área de un cuadrado se calcula elevando al cuadrado la medida de uno de sus lados.

La parte interesante de este problema es determinar el mínimo número de datos que hay que preguntar.

Análisis del caso

rea el conoci iento

An lisis e los ele entos el to o

Unidades de funcionalidad

ID	Unidad de funcionalidad	Prioridad	Obligatoriedad
1			

Interfaz

Algoritmo

Simulación

Anote aquí el comportamiento que espera de su programa.

Anote aquí la interfaz de salida.

3.- Caso 8: Determinando un descuento

Una tienda departamental ofrece descuentos directos sobre las compras que los clientes realizan.

El descuento normal va en función al tipo de cliente que la persona sea: A los clientes normales se les ofrece un 10 de descuento, y a los clientes premier un 15 de descuento.

Actualmente se tiene una promoción: en la compra de 5 artículos, se ofrecerá un descuento adicional al normal. Si el monto de la compra no excede de 1,000 pesos, se agrega un 5 de descuento; si el monto de la compra está entre 1,000 y 2,000 pesos, se agrega un descuento del 10 ; si la compra es mayor a 2,000 pesos, entonces el descuento adicional será del 15 .

En ese sentido, el menor descuento que se puede obtener es cuando un cliente normal compra menos de 1,000 pesos (15), y el máximo

descuento que se puede obtener es cuando un cliente premier compra más de 2,000 pesos (30).

El programa deberá recopilar la información de los precios de 5 productos, y determinar el monto de descuento aplicable.

Utilice arreglos.

Análisis del caso

rea el conoci iento

An lisis e los ele entos el to o

Unidades de funcionalidad

ID	Unidad de funcionalidad	Prioridad	Obligatoriedad
1			

Interfaz

Algoritmo

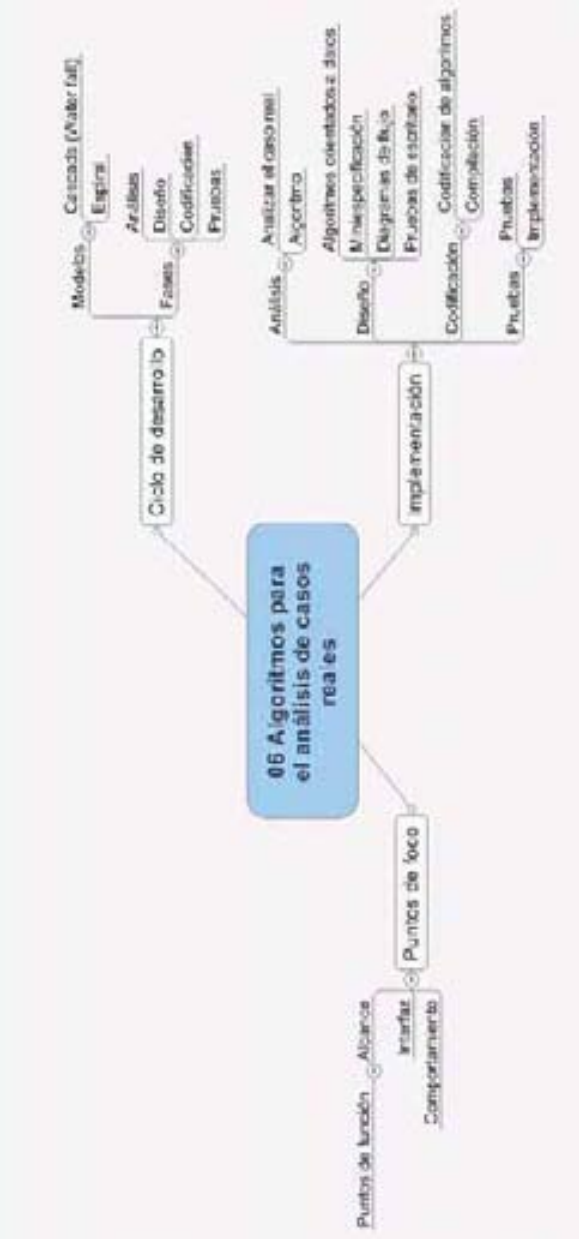
Simulación

Anote aquí el comportamiento que espera de su programa.

Anote aquí la interfaz de salida.

FIN DEL EJERCICIO ✱

Mapa mental del capítulo



Terminología

Alcance de un programa, 106
Algoritmo de programación, 109
Algoritmo, 109
Análisis, 102
Argot, 104
Ciclo de desarrollo, 95
Cliente, 103
Fase de análisis, 95
Fase de codificación, 96
Fase de diseño, 96
Fase de pruebas e implementación, 96
Función de programación, 95
Function points, 106
Information Technology Infraestructure Library, 103
Interfaz, 108
ITIL, 103
Modelo de Cascada, 95
Objetivo de una interfaz, 108
Prototipo, 109
Puntos de función, 106
Requerimientos del cliente, 103
Simulación, 109
Unidades de funcionalidad, 106
Waterfall model, 95

Preguntas

6.1 Existe un debate entre los analistas y los diseñadores de programas. Los analistas sostienen que el análisis es lo más importante, mientras que los diseñadores argumentan que la información que los analistas proporcionan nunca es suficiente, y que ellos terminan por corregir muchas deficiencias del análisis. ¿Usted a favor de quién se mostraría?

6.2 Siguiendo con el análisis anterior: si en efecto, los analistas entregan información insuficiente ¿Qué propone para que ello no siga sucediendo?

6.3 Algunos sostienen que la televisión está robándole a las personas su capacidad de asombro, y peor aún, su capacidad de análisis. Tal parece que la gente se está acostumbrando a que otras personas piensen por ellas. ¿Considera que las personas que más ven televisión tienen menos capacidad de análisis? Razone su respuesta.

6.4 ¿Cree que la capacidad de análisis es un carácter?

Notas:

Examen rápido

Parte 1 : Marque co n una cruz el inciso que considere que responde a la pregunta.

1) Consiste en una serie de pasos que nos llevan desde que analizamos una situación de la vida real, hasta que vemos ejecutarse, sin errores, el programa que automatiza las tareas.
(a) Función de compilación
(b) Análisis
(c) Proceso de desarrollo

2) Es la fase de más relevancia, dentro del proceso de desarrollo.
(a) Análisis
(b) Diseño
(c) Codificación

3) El código fuente del programa se origina en qué fase.
(a) Análisis
(b) Diseño
(c) Codificación

4) Los algoritmos son una herramienta de la fase de
(a) Análisis
(b) Diseño
(c) Codificación

5) Es una fase más importante para el cliente que para el desarrollador.
(a) Análisis
(b) Pruebas e implementación
(c) Diseño

Parte 2: Coloque en la línea la letra «F» si la sentencia es falsa, y «V» si la sentencia es verdadera.

6) El modelo de cascada es ideal para el diseño de prototipos.

7) La fase de diseño se especializa en definir cómo logrará satisfacer los requerimientos del diseño.

8) El usuario es el que paga por la tecnología, y no la gente de TI.

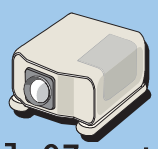
9) Es una mala práctica modificar los requerimientos del cliente con el fin de facilitar la labor de desarrollo.

10) Las actividades de desarrollo pueden ser manuales y automatizadas, pero son más las manuales.

www.detodoprogramacion.com

7

Algoritmos orientados a datos y miniespecificaciones



lp07.ppt

Objetivos: Aprenderá a elaborar algoritmos orientados a los datos y miniespecificaciones a partir de casos reales.

1. Aprenderá a transformar algoritmos a su versión orientada a los datos.
2. Aprenderá cuáles son las operaciones que son posibles realizar con los datos dentro de un programa.
3. Aprenderá el uso de estructuras de decisión y control en el desarrollo de procesos computarizados.
4. Aprenderá el concepto de los arreglos.
5. Aprenderá la forma en que se elaboran miniespecificaciones.
6. Aprenderá en qué consisten las pruebas de escritorio, y aprenderá a realizarlas.

www.detodoprogramacion.com

Contenido

Algoritmos orientados a los datos 136
Transformando algoritmos a su versión orientada a datos 138
Operaciones a nivel dato 139
Declaración 140
Cambio de estado 140
Muestra de datos 142
Estructuras de decisión y control 143
Estructuras de decisión 144
Contadores y acumuladores 145
Estructuras de control 146
Anidamiento 149
Arreglos 150
Miniespecificaciones 152
Cómo elaborar una miniespecificación 153
Miniespecificación de los casos prácticos 154
Caso 1: Mensaje 155
Caso 2: Niveles de servicio 156
Caso 3: Muestra de una tabla de multiplicar 157
Caso 4: Muestra las tablas de multiplicar del 1 al 5 159
Caso 5: Análisis de promedios 160
Pruebas de escritorio 163
Comprobando habilidades en el análisis de casos y el diseño de algoritmos orientados a datos 165
Mapa mental del capítulo 171
Terminología 172
Preguntas 172
Examen rápido 173

os datos no se crean ni se destruyen sólo cambian de estado.

Algoritmos orientados a los datos

Los algoritmos son muy útiles porque nos definen las actividades que se tienen que desarrollar para alcanzar un objetivo. Además nos definen el orden en que se tienen que desarrollar las actividades y el alcance general de las tareas a desarrollar.

La mente humana está tan habituada a colocar las cosas en secuencia, que los algoritmos son una forma natural de visualizar una tarea compuesta de muchas pequeñas tareas.

Aunque un algoritmo puede ser muy revelador para una persona, para la computadora una explicación descriptiva es bastante insuficiente; la computadora no entiende las cosas de forma descriptiva.

Imagine que queremos un programa que calcule el área de un cuadrado y que muestre el resultado del cálculo.

Para calcular el área de un cuadrado sabemos que basta con elevar al cuadrado la medida de uno de sus lados. Eso nosotros lo entendemos, pero la computadora no.

¿Es necesario que el problema especifique que la medida del lado es numérica, y que el resultado será también numérico? ¿Es necesario que el problema nos diga que la medida del lado es un número positivo? Por supuesto que no. Sin darnos cuenta nuestro conocimiento general simplemente lo sabe. Eso nosotros lo entendemos, pero la computadora no.

En términos generales, la computadora resulta bastante tonta, pues requiere que le digamos todo lo que tiene que hacer. A la computadora habría que decírselo de la siguiente manera.

Primero habría que decirle que el área de un cuadrado es igual a elevar al cuadrado la medida de uno de sus lados. Hasta ahí partimos de la misma base de conocimientos.

```
AreaCuadrado ← lado^2
```

Después habría que decirle que tanto el área calculada como la medida del lado son valores numéricos.

```
AreaCuadrado    d
lado            i
```

Después habría que decirle que la medida del lado no puede ser negativa.

```
lado            i[1-n]
```

Sólo hasta este punto la computadora será capaz de saber lo mismo que nosotros respecto al trabajo que se tiene que desarrollar.

En este capítulo daremos los primeros pasos para trasladar algo entendible por los humanos en algo entendible por la computadora.

Transformando algoritmos a su versión orientada a datos

Los *algoritmos orientados a datos* son los algoritmos que ponen en evidencia los datos manejados en cada una de las actividades a desarrollar, así como el dominio de estos datos.

Los algoritmos orientados a datos ya son documentos técnicos no orientados al usuario. Su finalidad verdadera es hacer del conocimiento del desarrollador los datos que estarán involucrados en el programa, así como sus respectivos dominios.

Siguiendo con el ejemplo del área del cuadrado, el algoritmo que podríamos discutir con el usuario quedaría como sigue.

1. Inicio.
2. Se pregunta la medida del lado.
3. Se calcula el área del cuadrado, elevando la medida del lado al cuadrado.
4. Se muestra el área del cuadrado.
5. Fin.

La regla para construir algoritmos orientados a los datos es muy sencilla. Partimos del supuesto que todos los datos manejados por un programa son expresiones. Sabemos que las expresiones sólo pueden manejarse en un programa si se cargan en memoria RAM, y sabemos que ninguna expresión puede cargarse en RAM si no se almacena en una variable.

La conclusión de todo esto es que cada dato manejado en el algoritmo terminará siendo una variable, y como toda variable, tendrá un tipo de datos específico, así como un dominio.

En nuestro ejemplo, en la actividad 2, se menciona lo siguiente.

2. Se pregunta la medida del lado.

Claramente podemos identificar un dato ahí, que es la medida del lado. Traduciendo esto en términos de variables, podemos decir que se trata de un dato entero, igual o mayor a uno, al cual le podemos dar el nombre lado.

Para poder representar esta información podemos recurrir a la siguiente notación.

(*ombre ariable*, *mbolo ipo* [*ominio*])

En ese sentido, la expresión quedaría como sigue.

(lado,i[1-n])

La expresión se deberá anotar en el texto del algoritmo cada vez que se mencione el dato en cuestión. Si en algún momento en el algoritmo se vuelve a mencionar el dato, sólo bastará mencionar el nombre de la variable entre paréntesis, dado que el dominio no cambiará en el transcurso del programa. La actividad que analizamos quedaría como sigue.

3. Se pregunta la medida del lado (lado,i[1-n]).

Nuestro algoritmo orientado a datos quedaría como sigue.

1. Inicio.
2. Se pregunta la medida del lado (lado,i[1-n]).
3. Se calcula el área del cuadrado (AreaCuadrado,d[1-n]), elevando la medida del lado al cuadrado (lado).
4. Se muestra el área del cuadrado (AreaCuadrado).
5. Fin.

El algoritmo orientado a datos revela los datos, y además abre todo un abanico de posibilidades. Cada dato que nos encontramos en un algoritmo orientado a datos tiene una razón de ser.

Nuestra función como desarrolladores es descubrir qué vamos a hacer con los datos en el programa. En caso de que no hagamos nada con ellos, probablemente no tiene sentido que formen parte del programa, dado que quitarlos no modifica el programa en cuanto a su efectividad.

Operaciones a nivel dato

Las *operaciones a nivel dato* son las operaciones que tienen como objetivo la declaración, cambio de estado y muestra de las variables.

a explicamos por qué todo dato termina siendo una variable. Las variables son llamadas así porque tienen la capacidad de variar el valor que contienen.

Declaración

Técnicamente decimos que una variable comienza a existir en el momento en que se declara. Entendemos como *declaración* cuando se le dice a la computadora que existe una referencia lógica a una posición física de la memoria RAM. Al declararse toda variable tiene un valor por omisión, que podremos ir cambiando a medida que el programa va haciendo cosas.

La declaración de una variable puede representarse simbólicamente de la siguiente manera.

```
(  ombre  ariable,  mbolo  ipo [  ominio ] )
```

Ejemplo.

```
(lado,i[1-n])
```

En esta representación sólo decimos que la variable existe, que es de tipo entero, y que podrá recibir valores iguales o mayores a 1.

Cambio de estado

Al valor que tiene una variable u objeto en un punto determinado del tiempo se le llama *estado*. Una variable cambia de estado cuando cambia su valor.

El cambio de valor se lleva a cabo mediante una operación de asignación.

Una *operación de asignación* es la operación a través de la cual se cambia el estado de una variable. En una operación de asignación pueden asignarse expresiones absolutas o expresiones compuestas, en las cuales la resolución de una expresión será el valor que se asigne a la variable.

Podemos representar la asignación de la siguiente manera.

```
 ombre  ariable ← E  presión
```

Ejemplo.

```
lado ← 10
```

En algunos lenguajes de programación sólo se puede realizar una asignación cuando la variable ha sido declarada previamente. Esto tiene sentido, dado que si no

hemos puesto a la computadora en el antecedente de la existencia de una variable, al usarla no sabrá qué hacer con ella. En la declaración anterior, por ejemplo, decimos que lado es igual a 10, sin embargo ¿es el dato de tipo byte, integer o decimal? La simple asignación no posee los elementos suficientes para inferir dicha información. Esa situación es tolerada por algunos lenguajes, que automáticamente asignan el valor cuyo dominio de tipo sea el más amplio; el programa funcionará sin marcar errores pero consumirá el máximo de memoria posible.

Algunos lenguajes también permiten realizar asignaciones al mismo tiempo en que se declara la variable. La forma simbólica en que podemos representar esto es la siguiente.

(*ombre ariable*, *mbolo ipo* [*ominio*]) ← *E presión*

Ejemplo.

(lado,i[1-n]) ← 10

En el mejor de los casos el valor que queremos asignar puede ser un valor absoluto o expresiones compuestas que están disponibles en un momento dado en nuestro programa; sin embargo, hay veces en que no podemos evitar preguntar al usuario por un dato que es requerido por el proceso, dado que no puede generarse de forma automática por los datos que conocemos.

Se realiza una *petición de datos* cuando un programa solicita al usuario datos que el proceso requiere y que no es posible obtener de otra manera.

La representación simbólica de una petición de datos es la siguiente.

» *ombre ariable*

Ejemplo.

» lado

En este ejemplo estaríamos solicitando al usuario el valor que queremos asignar a la variable lado.

En una petición de datos se utiliza el *s mbolo de petición* ( ) que consiste en una punta de flecha doble apuntando hacia la derecha. Después del símbolo debe ir el nombre de la variable que ha de adquirir el valor que se requiere al usuario.

Muestra de datos

La *muestra de datos* es la acción de mostrar una expresión a través de la interfaz.

Con generalidad queremos que el programa se comunique con nosotros, y una forma efectiva es la salida en pantalla, aunque también puede ser en otros medios como el papel.

La forma en que se representa simbólicamente la muestra de datos es la siguiente.

```
E presión
```

Ejemplo.

```
« lado
```

En nuestro ejemplo, el programa nos comunicaría el contenido de la variable lado. Al momento de mostrar datos es posible construir expresiones complejas con el fin de adornar la salida. Vea el siguiente ejemplo.

```
« "El valor del lado es: " + lado
```

Si el valor de la variable lado es de 20, el mensaje que aparecería como salida sería el siguiente.

```
El valor del lado es: 20
```

Incluso al momento de mostrar datos una expresión compleja puede realizar procesamiento. Vea el siguiente ejemplo.

```
« "Si el lado vale " + lado + " entonces el área del cuadrado es " + (lado  2)
```

Si el valor de la variable lado es de 20, el mensaje que aparecería como salida sería el siguiente.

```
Si el lado vale 20 entonces el área del cuadrado es 400
```

No se recomienda realizar procesamiento al momento de mostrar los datos. Lo ideal es realizar cálculos sobre las variables, y luego formular una expresión compleja que no involucre resolución de cálculos.

En una muestra de datos se utiliza el *s mbolo de muestra de datos* ( ) que consiste en una punta de flecha doble apuntando hacia la izquierda. Después del símbolo debe ir la expresión a mostrar.

Como el símbolo de petición y el símbolo de muestra de datos son muy similares, acuérdese que leemos de izquierda a derecha. Ahora coloque al usuario primero (del lado izquierdo) y a la computadora después (del lado derecho), y ahora vea quién le da datos a quién. La figura 7.1 ilustra lo que acabamos de sugerir.

Figura 0 .01

Diferenciando el s mbolo de petición de muestra de datos.

Estructuras de decisión y control

Las estructuras de decisión y control son aquellas que utilizan variables pero no afectan el estado de una variable existente, sino que permiten el uso de una o más variables en la composición de una expresión lógica que determina la ejecución de bloques de código dentro de un programa, ya sea de forma única o reiterada.

Estas estructuras siempre tienen líneas que indican el inicio y el final de la estructura. A estas líneas se les conoce como *delimitadores de la estructura*, que juegan un rol de inicio y fin, siendo llamados por ello *delimitador de inicio* y *delimitador de fin*, respectivamente. Al encontrar un delimitador de inicio, decimos que se está *iniciando una estructura*; al encontrar un delimitador de fin, decimos que se *finaliza una estructura*.

Entre el inicio y el fin de una estructura siempre habrá código. A este bloque de código cuya ejecución dependerá del valor de una expresión lógica evaluada por la estructura se le conoce como *código controlado*. Por simple representación se

aplica *sangr a* al código controlado, es decir el código controlado siempre se escribe más a la derecha que los delimitadores de la estructura.

Simbólicamente hablaríamos de lo siguiente.

```
elimitador de inicio
        ódigo controlado
elimitador de fin
```

Estructuras de decisión

Se le llama *estructura de decisión* a la estructura de código en la cual una expresión lógica determina la ejecución por única vez de un bloque de código. La más clásica de las estructuras de decisión es la *condicional*.

La condicional puede ser de dos tipos: condicional simple y condicional compuesta.

Condicional si mple. La *condicional simple* es aquella en la que sólo especificamos el bloque de código que deseamos que se ejecute si la expresión lógica de la operación de comparación resuelve por verdadero (True). La condicional simple se representa simbólicamente de la siguiente forma.

```
Si E  presión  ógica Entonces
        Parte del algoritmo a ejecutar si la E  presión  ógica es  rue
Fin Si
```

Ejemplo.

```
Si lado >= 1 Entonces
        AreaCuadrado ← lado  2
Fin Si
```

En este ejemplo sólo permitimos que se calcule el área del cuadrado si el valor almacenado en la variable lado es mayor o igual a uno.

Condicional compuesta. La *condicional compuesta* es aquella en la además de especificar el bloque de código que deseamos que se ejecute si la expresión lógica de la operación de comparación resuelve por verdadero (True), especificamos también el bloque de código que deseamos que se ejecute si la expresión lógica de la

operación de comparación resuelve por falso (False). La condicional compuesta se representa simbólicamente de la siguiente forma.

```
Si E  presión  ógica Entonces
        Parte del algoritmo a ejecutar si la E  presión  ógica es  rue
De lo contrario
        Parte del algoritmo a ejecutar si la E  presión  ógica es  alse
Fin Si
```

Ejemplo.

```
Si lado >= 1 Entonces
        AreaCuadrado ← lado  2
De lo contrario
        » "El lado no puede ser menor a uno"
Fin Si
```

Contadores y acumuladores

Antes de analizar las estructuras de control es necesario que entendamos dos técnicas de programación básica: los contadores y los acumuladores.

Contadores. Los *contadores* son técnicas de cambio de estado en donde una variable tiene la habilidad de contar las veces en que su estado cambia. A grandes rasgos, un contador es el resultado de sumarle 1 a una variable, asignando el resultado a la misma variable. Su representación simbólica sería como sigue.

```
 ariable ←  ariable + 1
```

Ejemplo.

```
i ← i + 1
i ← i + 1
i ← i + 1
i ← i + 1
i ← i + 1
```

En este ejemplo, si la expresión i ← i + 1 se ejecutara 5 veces, el valor sería 5. Es importante comentar que muchos procesos requieren inicializar los contadores antes de utilizarlos.

Un contador es muy útil en el sentido que permite conocer el número de veces que un programa ha pasado por una determinada línea.

Acumuladores. Los *acumuladores* son técnicas de cambio de estado en donde una variable tiene la habilidad de acumular un valor al valor que ya posee. Su representación simbólica sería como sigue.

ariable ← ariable + E presión

Ejemplo.

```
i ← i + 10
i ← i + 2
i ← i + 5
i ← i + 3
```

En este ejemplo, i nunca pierde su valor. Al finalizar el proceso anterior, i vale 20, que es el resultado de haber acumulado todos los valores proporcionados.

Un acumulador es muy útil en el sentido que permite realizar sumatorias de valores.

Estructuras de control

Se le llama *estructura de control* a la estructura de código en la cual una expresión lógica determina la ejecución reiterada de un bloque de código. La más clásica de las estructuras de control es el *bucle* (*loop*).

Al bloque de código que se ejecuta dependiendo del valor de una expresión lógica en un bucle se le llama código controlado.

Un bucle puede ser de dos tipos: bucle de comparación al inicio y bucle de comparación al final.

Bucle de comparación al inicio. Un *bucle de comparación al inicio* es aquél en el cual la expresión lógica se resuelve al inicio del proceso, de tal forma que el código controlado puede ejecutarse una, varias o ninguna vez. En un bucle de comparación al inicio, el código controlado se ejecutará siempre y cuando la expresión lógica sea verdadera (True). Su representación simbólica es como sigue.

```
 nicialización del contador
Mientras E  presión  ógica Entonces
            so de contador
          Parte del algoritmo a ejecutar si la E  presión  ógica es   rue
Fin Mientras
```

Ejemplo.

```
i ← 0
Mientras i <= 5 Entonces
          i ← i + 1
          « "El valor de i es : " + i
Fin Mientras
```

El ejemplo representa un bucle que estará repitiendo un proceso consistente en mostrar un mensaje. El resultado del ejemplo es el siguiente.

```
El valor de i es: 1
El valor de i es: 2
El valor de i es: 3
El valor de i es: 4
El valor de i es: 5
```

Vea cómo el ejemplo incluye un contador en la línea adyacente a la línea en donde se encuentra la condición a evaluar. Sin ese contador el ciclo sería infinito, dado que la expresión lógica de comparación nunca dejaría de ser verdadera (True). Este bucle, al llegar al final de la estructura, traslada el control del proceso al inicio, evaluando si la expresión lógica sigue siendo verdadera.

Después de haber impreso el mensaje correspondiente al número 5, el contador se incrementa. Como i ya no es menor o igual a 5, el proceso ya no se ejecutará más.

¿Cuál será la salida si modificamos la inicialización del contador?

```
i ← 10
Mientras i <= 5 Entonces
        i ← i + 1
        « "El valor de i es : " + i
Fin Mientras
```

La respuesta es que no tendrá salida, dado que desde el inicio la expresión lógica fue falsa (False).

Bucle de comparación al final. Un *bucle de comparación al final* es aquél en el cual la expresión lógica se resuelve al final del proceso, de tal forma que el código controlado puede ejecutarse al menos una vez, y luego varias.

La diferencia respecto al bucle de comparación al inicio, es que este bucle se ejecutará al menos una vez. En un bucle de comparación al final, el código controlado se ejecutará al menos una vez, y de la segunda vez en adelante, se ejecutará siempre y cuando la expresión lógica sea verdadera (True). Su representación simbólica es como sigue.

```
 nicialización del contador
Inicio
        Parte del algoritmo a ejecutar si la E  presión  ógica es   rue
          so de contador
Hasta que E  presión  ógica
```

Ejemplo.

```
i ← 1
Inicio
        « "El valor de i es: " + i
        i ← i + 1
Hasta que i > 5
```

El resultado es el mismo que si se utilizara el bucle de comparación al inicio.

¿Cuál será la salida si modificamos la inicialización del contador?

```
i ← 10
Inicio
        « "El valor de i es : " + i
        i ← i + 1
Hasta que i > 5
```

La respuesta sería la siguiente.

```
El valor de i es: 10
```

Al menos se ejecuta el proceso una vez. El programa es incapaz de saber que la expresión lógica resolverá por verdadero (True), dado que aún no llega a esa parte del programa. Al llegar a la línea que posee la expresión lógica detecta que resuelve por verdadero, por lo que el código controlado ya no se vuelve a ejecutar.

Otro cambio sustancial es que en el bucle de comparación inicial se establece que el código controlado se ejecute mientras una condición se cumpla por verdadero; en el caso del bucle de comparación al final el código controlado se ejecuta hasta que se cumpla por verdadero la condición, por lo cual al adquirir el valor de verdadero el ciclo concluye. Por esa razón las condiciones que aplican a un caso y a otro son opuestas en cierto sentido (i <=5, i > 5); mientras que en uno el valor verdadero provoca la ejecución, en otro la da por terminada.

Al igual que en el bucle de comparación al inicio, el contador se incluye adyacente a la línea que posee la condición a evaluar. El valor del contador al inicializarse, en el caso de los bucles de comparación al inicio, se establece uno antes del primer elemento relevante (0), mientras que en el caso de los bucles de comparación al final se establece con el valor del primer elemento relevante (1).

Anidamiento

Se conoce como *anidamiento* a la acción de que una estructura de decisión y control forme parte del código controlado de otra estructura.

Esto quiere decir que dentro de un bucle puede haber una condicional, y dentro de esa condicional puede haber un bucle, y así. En realidad no hay límites en el anidamiento. Lo que sí existe son las siguientes reglas que deberán respetarse al anidar estructuras.

- La última estructura en iniciarse debe ser la primera en finalizarse.
- Debe haber el mismo número de delimitadores de inicio que de fin.
- No deben finalizarse estructuras que no se han iniciado.
- No deben dejarse estructuras iniciadas sin finalizarse.

Este ejemplo de anidamiento muestra cómo puede haber un condicional dentro de un bucle. En el ejemplo nos estamos asegurando que el contador, al inicio del proceso desarrollado por el bucle, siempre sea uno.

```
Inicio
        Si i <> 1 Entonces
                i ← 1
        Fin Si
        « "El valor de i es : " + i
        i ← i + 1
Hasta que i > 5
```

Vea cómo la condicional forma parte del código controlado del bucle. Aún y cuando olvidáramos inicializar en 1 al contador i, el programa es capaz de corregir el problema en tiempo de ejecución.

Arreglos

Los *arreglos* son variables que permiten asociar, a través de un mismo nombre de variables, varias posiciones de memoria.

Esto es muy útil cuando se desea almacenar un conjunto de datos de una misma naturaleza, sin tener que recurrir a nombres diversos de variables.

Aunque unidas por un mismo nombre, las posiciones de memoria manejadas en forma de arreglo se diferencian por un número único al que conocemos como *sub ndice*. El subíndice es un número entero no negativo. Dependiendo del lenguaje, puede iniciar desde cero o desde uno.

A cada posición de memoria almacenada en el arreglo se le llama *elemento* (*item*).

Para entender la utilidad de los arreglos, imagine que desea almacenar los gastos del primer semestre del año. Sin arreglos, la solución sería como sigue.

```
GastosEnero ← 10000
GastosFebrero ← 8000
GastosMarzo ← 4000
GastosAbril ← 11000
GastosMayo ← 8500
GastosJunio ← 7000
```

Ahora imagine que no son sólo 6 meses, sino que son todos los meses de 5 años. Daría un total de 60 meses, por lo que tendría que inventar 60 diferentes nombres de variable, lo cual se presta mucho a los errores. Si además le pidieran que sumara todos los gastos a partir de las variables, construir la expresión de suma sería un problema.

Afortunadamente existen los arreglos.

La representación simbólica para la declaración de un arreglo es la siguiente.

(*ombre ariable*(*Elementos el rreglo*), *mbolo ipo* [*ominio*])

Ejemplo.

(GastosMensuales(6),i[1-n])

Lo que indicaría que el arreglo se llamaría GastosMensuales, que tendrá 6 elementos, y que almacena datos enteros mayores o iguales a uno.

Para referir a un elemento determinado del arreglo se utiliza el subíndice del elemento. Referir un elemento nos permite extraer el dato que almacena o realizar asignaciones. La representación simbólica de un elemento es la siguiente.

ombre ariable(*ub ndice*)

Ejemplo.

GastosMensuales(3)

Siguiendo con nuestro ejemplo, cargar valores en el arreglo sería como sigue.

GastosMensuales(1) ← 10000

GastosMensuales(2) ← 8000

GastosMensuales(3) ← 4000

GastosMensuales(4) ← 11000

GastosMensuales(5) ← 8500

GastosMensuales(6) ← 7000

En caso de que fueran 60 los elementos del arreglo ¿cómo podríamos ver la información almacenada sin tener que escribir 60 líneas? Sencillo.

```
i ← 0
Mientras i <= 60 Entonces
        i ← i + 1
        « "El valor del elemento " + i  + " es : " + GastosMensuales(i)
Fin Mientras
```

Miniespecificaciones

A este momento ya casi sabe miniespecificar, dado que ha aprendido las representaciones simbólicas de las operaciones posibles con los datos.

Se conoce como *miniespecificación* a la representación abstracta y simbólica entendible por el ser humano, que muestra al máximo detalle las operaciones que un programa debe hacer con los datos.

Una miniespecificación es una de las herramientas clave para la especificación sistemática de procesos computarizados.

Algunas de las características de las miniespecificaciones son las siguientes.

1. Debe tener un inicio y un fin.
2. Debe tener líneas numeradas.
3. Debe ser independiente de los lenguajes de programación y las tecnologías.
4. Debe contener un cuadro básico de instrucciones que puedan ser representadas en cualquier lenguaje.

. Deben describirse mayormente a través de representaciones simbólicas.

6. Deben incluir todos los datos del algoritmo orientado a datos, en forma de declaraciones de memoria.

. En caso de funcionalidad que no cuente con simbología permitida, deberá describirse textualmente lo que se desea hacer (extraer datos de una base de datos o instanciar objetos especiales, por ejemplo).

Una miniespecificación básica sería la siguiente.

```
1   INICIO:
2           iniespecificación
3   :FIN
```

Las miniespecificaciones permiten especificar bloques de código, no aplicaciones enteras. Por esa razón son pequeñas, de tal forma que describan al desarrollador las tareas específicas que debe codificar.

Cómo elaborar una miniespecificación

Todo parte del algoritmo orientado a datos, que es el que provee la información, tanto de las actividades que es necesario realizar como los datos utilizados.

Veamos nuestro ejercicio de cálculo del área de cuadrado.

1. Inicio.
2. Se pregunta la medida del lado (lado,i[1-n]).
3. Se calcula el área del cuadrado (AreaCuadrado,d[1-n]), elevando la medida del lado al cuadrado (lado).
4. Se muestra el área del cuadrado (AreaCuadrado).
5. Fin.

El primer paso es crear el marco general de la miniespecificación.

```
1   INICIO:
2           iniespecificación
3   :FIN
```

Posteriormente se declaran como variables todos los datos utilizados en el algoritmo.

```
1   INICIO:
2           (lado,i[1-n])
3           (AreaCuadrado,d[1,n])
4   :FIN
```

a que se tienen los datos declarados, se sigue el algoritmo y se traducen simbólicamente las acciones.

1. Los cálculos se traducen en asignaciones.
2. Las decisiones se traducen en condicionales. Dependiendo la relevancia que se le de al resultado por verdadero o por falso, se decidirá el uso de condicional simple o compuesta.
3. Los cálculos reiterativos se traducen en estructuras de control.

4. Los datos que no se tienen se traducen en peticiones de datos. Generalmente aquí se acompaña la petición con un mensaje de petición que solicita el dato al usuario.

. Los resultados y mensajes se traducen en muestras de datos.

En nuestro ejemplo las traducciones simbólicas ocurren como sigue.

Algoritmo orientado a datos	Traducción simbólica
2. Se pregunta la medida del lado (lado,i[1-n]).	« "Cuál es la medida del lado: " » lado
3. Se calcula el área del cuadrado (AreaCuadrado,d[1-n]), elevando la medida del lado al cuadrado (lado).	AreaCuadrado ← lado^2
4. Se muestra el área del cuadrado.	« "El área es " + AreaCuadrado

Finalmente, nuestra miniespecificación quedaría como sigue.

```
1   INICIO:
2           (lado,i[1-n])
3           (AreaCuadrado,d[1,n])
4
5           « "Cuál es la medida del lado: "
6
7           » lado
8           AreaCuadrado ← lado^2
9           « "El área es " + AreaCuadrado
    :FIN
```

Con esta miniespecificación sería muy sencillo producir un programa en cualquier lenguaje.

Miniespecificación de los casos prácticos

En esta parte se transformarán los algoritmos de los casos prácticos, desarrollados en el capítulo anterior, a fin de orientarlos a los datos.

Una vez que se tenga el algoritmo orientado a datos, se procederá a crear la miniespecificación de los programas. Por regla general se declararán todas las variables.

Al final podrá comprobar sus habilidades con el ejercicio que muestra los programas complementarios.

Caso 1: Mensaje

Este programa tiene como finalidad comprobar técnicas y herramientas de desarrollo. Lo que el programa debe hacer es preguntar por un mensaje de saludo, para posteriormente mostrar el mensaje que se le proporcionó.

Este ejercicio no ofrece problemas en la selección de símbolos, dado que es muy evidente que se trata de una petición y una muestra de datos. El único detalle es que la petición del mensaje implica una muestra de datos (texto que solicita el mensaje), así como el acceso de los datos.

Algoritmo

1. Inicio.
2. Se pregunta un mensaje.
3. Se muestra el mensaje.
4. Fin.

Algoritmo orientado a datos

1. Inicio.
2. Se pregunta un mensaje (Mensaje, x(30)) .
3. Se muestra el mensaje (Mensaje).
4. Fin.

Miniespecificación

```
1   INICIO:
2           (Mensaje, x(30))
3           « "Escribe un mensaje a mostrar: "
4           » Mensaje
5           « Mensaje
6   :FIN
```

Caso 2: Niveles de servicio

Se desea saber la disponibilidad del servicio de Internet para una persona, expresado en horas de disponibilidad.

Dicho nivel de servicio depende del grado de disponibilidad que ofrece la computadora que utiliza, así como del grado de disponibilidad del enlace a Internet que utiliza la máquina.

El horario de labores de la persona que utiliza la computadora y el enlace a Internet es de 8 horas.

Para poder realizar los cálculos adecuadamente, tome en cuenta que la disponibilidad de un sistema de dispositivos (dos o más dispositivos que trabajan de forma conjunta) se calcula multiplicando los porcentajes de grado de disponibilidad de cada uno de ellos, y el resultado siempre será menor a la disponibilidad del menor.

Para calcular el tiempo de disponibilidad de un servicio (en este caso el acceso a la información publicada en Internet), sólo basta con multiplicar el tiempo en que es requerido el servicio por la disponibilidad del sistema de dispositivos.

Este ejemplo tiene el mismo comportamiento que el ejercicio anterior. La única diferencia es que involucra actividades de cambio de estado que ayudan al cálculo de la información que el caso pretende (Disponibilidad de servicio y tiempo de disponibilidad).

Algoritmo

1. Inicio.
2. Se pregunta la disponibilidad de la computadora.
3. Se pregunta la disponibilidad del enlace.
4. Se pregunta el tiempo laborable en horas.
5. Se calcula la disponibilidad del sistema de dispositivos.
6. Se calcula el tiempo de disponibilidad del servicio.
7. Se muestra el tiempo de disponibilidad del servicio.
8. Fin.

Algoritmo orientado a datos

1. Inicio.
2. Se pregunta la disponibilidad de la computadora (DC,d[0-1]).
3. Se pregunta la disponibilidad del enlace (DE,d[0-1]).
4. Se pregunta el tiempo laborable en horas (TL,d[0-1]).

5. Se calcula la disponibilidad del sistema de dispositivos (DS,d[0-1]).
6. Se calcula el tiempo de disponibilidad del servicio (DH,d[0-1]).
7. Se muestra el tiempo de disponibilidad del servicio (DH).
8. Fin.

Miniespecificación

```
INICIO:
        (DC,d[0-1])
        (DE,d[0-1])
        (TL,d[0-1])
        (DS,d[0-1])
        (DH,d[0-1])
        « "Grado de disponibilidad de la computadora:"
        » DC
        « "Grado de disponibilidad del enlace:"
        » DE
        « "Tiempo laborable en horas:"
        » TL
        DS ← DC * DE
        DH ← DS * TL
        « "La disponibilidad del servicio de Internet es de " + DH + " horas"
:FIN
```

Caso 3: Muestra de una tabla de multiplicar

El programa debe mostrar cualquier tabla de multiplicar que le solicitemos. La única restricción es que la tabla deberá ser de un número mayor o igual a 1, y menor o igual a 10.

Si se pide una tabla de un número que no esté entre el 1 y el 10, el programa debe indicar, mediante un mensaje, que no puede continuar.

Este ejemplo agrega elementos complejos. Además de requerir de petición y muestra de datos, así como cambios de estado, requiere de condicionales y estructuras de control. Al hacer referencias a sentencias "Si....Entonces...", es necesario aplicar una condicional; en virtud de que se desean hacer cosas tanto si la condición se cumple (True) como si no (False), es necesario que la condicional sea compuesta.

Otro punto complejo es que solicita que se muestre una tabla de multiplicar. Todos conocemos cómo son las tablas de multiplicar, y hay dos formas de generarlas: no usando estructuras de control, y usándolas.

Se explorarán las dos opciones en forma de miniespecificaciones.

Algoritmo

1. Inicio.
2. Se pregunta qué tabla se desea.
3. Si la tabla está entre 1 y 10, entonces se imprime la tabla solicitada.
4. Si la tabla no está entre el 1 y 10, entonces se muestra un mensaje de error.
5. Fin.

Algoritmo orientado a datos

1. Inicio.
2. Se pregunta qué tabla se desea (Tabla,i[1-10]).
3. Si la tabla (Tabla) está entre 1 y 10, entonces se imprime la tabla solicitada.
4. Si la tabla no está entre el 1 y 10, entonces se muestra un mensaje de error.
5. Fin.

Miniespecificación (sin utilizar estructuras de control)

```
1   INICIO:
2           (Tabla,i[1-10])
3           « "Qué tabla deseas: "
4           » Tabla
5           Si Tabla >= 1 ∧ Tabla <=10 Entonces
6                   « Tabla + " X" + 1 + "=" + Tabla *1
7                   « Tabla + " X" + 2 + "=" + Tabla *2
8                   « Tabla + " X" + 3 + "=" + Tabla *3
9                   « Tabla + " X" + 4 + "=" + Tabla *4
10                  « Tabla + " X" + 5 + "=" + Tabla *5
11                  « Tabla + " X" + 6 + "=" + Tabla *6
12                  « Tabla + " X" + 7 + "=" + Tabla *7
13                  « Tabla + " X" + 8 + "=" + Tabla *8
14                  « Tabla + " X" + 9 + "=" + Tabla *9
15                  « Tabla + " X" + 10 + "=" + Tabla *10
16          De lo contrario
17                  « "Válido sólo del 1 al 10"
18          Fin Si
19  :FIN
```

Miniespecificación (usando estructuras de control)

```
1   INICIO:
2           (i,i[0-n])
3           (Tabla,i[1-10])
4           « "Qué tabla deseas: "
5           » Tabla
6           Si Tabla >= 1 ∧ Tabla <=10 Entonces
7                   i ← 1
8                   Inicio
9                           « Tabla + " X" + i + "=" + Tabla * i
10                          i ← i + 1
11                  Hasta que i > 10
12          De lo contrario
13                  « "Válido sólo del 1 al 10"
14          Fin Si
    :FIN
```

La opción utilizando estructuras de control debe preferirse. La primera alternativa, aunque más simple, crece en complejidad dependiendo del número de resultados que se deseen. Si en lugar de la tabla del 1 al 10 nos pidieran la tabla del 1 al 1000, al primer programa habría que agregarle 990 líneas más, mientras que al segundo sólo sería necesario cambiarle la condición de la estructura de control, de i > 10 a i > 1000, siendo totalmente funcional.

No sólo piense en la simplicidad a corto plazo. Piense en el mantenimiento de los programas a futuro. Lo que parece cómodo hoy puede causarnos noches de desvelo mañana; intente hacer sus programas escalables, a fin de que crezcan en su capacidad de procesamiento con el mínimo de cambios.

Caso 4: Muestra las tablas de multiplicar del 1 al 5

El programa debe mostrar todas las tablas de multiplicar del 1 al 5, incluyendo un encabezado por tabla.

Este es un extraño caso en donde el algoritmo orientado a datos no es relevante. La razón de ello es que se trata de cálculo de tablas de multiplicar que todos sabemos cómo se calculan. Quizá es más relevante saber que la forma más práctica de hacer este programa es anidando dos estructuras de control; los contadores utilizados por las estructuras es todo lo que necesitamos, por lo que no es necesario ningún cuestionamiento.

Si el programa puede proveerse a sí mismo de datos, no es necesario que el algoritmo orientado a datos manifieste datos. Un ejemplo de esto es cuando hacemos un programa que muestra la hora del sistema: no es necesario preguntarla ni declararla, dado que el sistema ya sabe eso. Recuerde la regla general: el menor número de datos, el mayor número de resultados.

Algoritmo orientado a datos (es igual al Algoritmo)

1. Inicio.
2. Muestra el encabezado de la tabla del 1.
3. Se calcula la tabla del 1 y se muestra.
4. Muestra el encabezado de la tabla del 2.
5. Se calcula la tabla del 2 y se muestra.
6. Muestra el encabezado de la tabla del 3.
7. Se calcula la tabla del 3 y se muestra.
8. Muestra el encabezado de la tabla del 4.
9. Se calcula la tabla del 4 y se muestra.
10. Muestra el encabezado de la tabla del 5.
11. Se calcula la tabla del 5 y se muestra.
12. Fin.

Miniespecificación

```
1   INICIO:
2           (i,i[0-n])
3           (y,i[0-n])
4           i ← 1
5           Inicio
6                   « "TABLA: " + i
7                   y ← 1
8                   Inicio
9                           « i + " X" + y + "=" + i * y
10                          y ← y + 1
11                  Hasta que y > 10
12                  i ← i + 1
13          Hasta que i > 5
14  :FIN
```

Caso 5: Análisis de promedios

El programa debe pedir 5 números, para después mostrarlos indicando si están o no por encima de su promedio. Al final deberá enviarse un resumen en donde se indique la suma total de los números, y el promedio calculado.

Para la petición de datos utilice un arreglo.

El punto interesante de este programa es que utiliza acumuladores y arreglos. Al igual que el ejercicio de la tabla de multiplicar, se puede hacer utilizando estructuras

de control o no utilizándolas; exploraremos ambos casos aunque la recomendación sea la misma: use las estructuras de decisión y control cuando le reduzcan el trabajo a largo plazo, y evada el estilo de programar en el que la menor variación nos lleva a tener que codificar mucho.

Es interesante ver cómo un contador puede ser reutilizado.

Algoritmo

1. Inicio.
2. Se pregunta el valor del elemento 1.
3. Se pregunta el valor del elemento 2.
4. Se pregunta el valor del elemento 3.
5. Se pregunta el valor del elemento 4.
6. Se pregunta el valor del elemento 5.
7. Se realiza la suma de todos los elementos.
8. Se calcula el promedio de la serie de números.
9. Se muestra la lista de valores, informando si el número se encuentra por encima o no del promedio.
10. Se muestra el resultado de la suma.
11. Se muestra el resultado del promedio.
12. Fin.

Algoritmo orientado a datos

1. Inicio.
2. Se pregunta el valor del elemento 1 (Entrada(1),i[0-n]).
3. Se pregunta el valor del elemento 2 (Entrada(2),i[0-n]).
4. Se pregunta el valor del elemento 3 (Entrada(3),i[0-n]).
5. Se pregunta el valor del elemento 4 (Entrada(4),i[0-n]).
6. Se pregunta el valor del elemento 5 (Entrada(5),i[0-n]).
7. Se realiza la suma de todos los elementos (Suma,d[0-n]).
8. Se calcula el promedio de la serie de números (Promedio,d[0-n]).
9. Se muestra la lista de valores (Entrada), informando si el número (Entrada(n))se encuentra por encima o no del promedio (Promedio).
10. Se muestra el resultado de la suma (Suma).
11. Se muestra el resultado del promedio (Promedio).
12. Fin.

www.detodoprogramacion.com

Miniespecificación (sin usar estructuras)

```
1   INICIO:
2           (Entrada(5),i[0-n])
3           (Suma,d[0-n])
4           (Promedio,d[0-n])
5           Suma ← 0
6           « "Valor para el elemento 1:"
7           » Entrada(1)
8           « "Valor para el elemento 2:"
9           » Entrada(2)
10          « "Valor para el elemento 3:"
11          » Entrada(3)
12          « "Valor para el elemento 4:"
13          » Entrada(4)
14          « "Valor para el elemento 5:"
15          » Entrada(5)
16          Suma ← Entrada(1) + Entrada(2) + Entrada(3) + Entrada(4) + Entrada(5)
17          Promedio ← Suma / 5
18          Si Entrada(1) >= Promedio Entonces
19              « "Número " + Entrada(1) + " mayor o igual al promedio: "
20          De lo contrario
21              « "Número " + Entrada(1) + " menor al promedio: "
22          Fin Si
23          Si Entrada(2) >= Promedio Entonces
24              « "Número " + Entrada(2) + " mayor o igual al promedio: "
25          De lo contrario
26              « "Número " + Entrada(2) + " menor al promedio: "
27          Fin Si
28          Si Entrada(3) >= Promedio Entonces
29              « "Número " + Entrada(3) + " mayor o igual al promedio: "
30          De lo contrario
31              « "Número " + Entrada(3) + " menor al promedio: "
32          Fin Si
33          Si Entrada(4) >= Promedio Entonces
34              « "Número " + Entrada(4) + " mayor o igual al promedio: "
35          De lo contrario
36              « "Número " + Entrada(4) + " menor al promedio: "
37          Fin Si
38          Si Entrada(5) >= Promedio Entonces
39              « "Número " + Entrada(5) + " mayor o igual al promedio: "
40          De lo contrario
41              « "Número " + Entrada(5) + " menor al promedio: "
42          Fin Si
43          « "Suma: " + Suma
44          « "Promedio: " + Promedio
45  :FIN
```

Miniespecificación (usando estructuras)

```
1   INICIO:
2           (i,i[0-n])
3           (Entrada(5),i[0-n])
4           (Suma,d[0-n])
5           (Promedio,d[0-n])
6           Suma ← 0
7           i ← 1
8           Inicio
9                   « "Valor para el elemento " + i + ": "
10                  » Entrada(i)
11                  Suma ← Suma + Entrada(i)
12                  i ← i + 1
13          Hasta que i > 5
14          Promedio ← Suma / 5
15          i ← 1
16          Inicio
17                  Si Entrada(i) >= Promedio Entonces
18                      « "Número " + Entrada(i) + " mayor o igual al promedio: "
19                  De lo contrario
20                      « "Número " + Entrada(i) + " menor al promedio: "
21                  Fin Si
22                  i ← i + 1
23          Hasta que i > 5
24          « "Suma: " + Suma
25          « "Promedio: " + Promedio
26  :FIN
```

Pruebas de escritorio

Las *pruebas de escritorio* son pruebas manuales que se encargan de visualizar el comportamiento de los estados de las variables en el transcurso de la ejecución de un programa.

La forma de hacer una prueba de escritorio es muy sencilla: se define una tabla de dos columnas; en la primer columna se colocan las variables, y en la segunda se colocan los valores que las variables adquieren en un proceso.

Deberá simularse línea por línea la ejecución del programa. En cada línea donde exista un cambio de estado o una petición de información, se deberá actualizar la columna de valores de la prueba.

Tomemos como muestra el caso del análisis de promedio, tomando la miniespecificación que no tiene estructuras por ser la más secuencial.

Consideremos como valores ingresados 10, 15, 25, 9, y 4.

A la línea 5, la tabla luciría así.

Variable	Valor
Entrada	
(1)	
(2)	
(3)	
(4)	
(5)	
Suma	0
Promedio	

A la línea 15, la tabla luciría así.

Variable	Valor
Entrada	
(1)	10
(2)	15
(3)	25
(4)	9
(5)	4
Suma	0
Promedio	

A la línea 16, la tabla luciría así.

Variable	Valor
Entrada	
(1)	10
(2)	15
(3)	25
(4)	9
(5)	4
Suma	63
Promedio	

A la línea 17, la tabla luciría así.

Variable	Valor
Entrada	
(1)	10
(2)	15
(3)	25
(4)	9
(5)	4
Suma	63
Promedio	12.60000

La prueba de escritorio es muy útil cuando los procesos no resuelven correctamente un cálculo. Si las operaciones están bien estructuradas, pero el resultado está mal,

lo más probable es que uno o varios de los datos no está adquiriendo los valores de forma adecuada. La prueba de escritorio ayuda a ver las omisiones y cálculos erróneos, incluso sin necesidad de ejecutar los programas en un esquema a prueba y error.

Afortunadamente hay herramientas de desarrollo que incluyen ya pruebas de escritorio automatizadas, en las que podemos ver los valores que adquieren las variables en forma dinámica.

Ejercicio 07.01

Comprobando habilidades en el análisis de casos y el diseño de algoritmos orientados a datos

Elabore lo siguiente para cada uno de los casos que se propongan.

- ❒ Algoritmo orientado a datos.
- ❒ Miniespecificación.

1.- Caso 6: Regla de tres simple

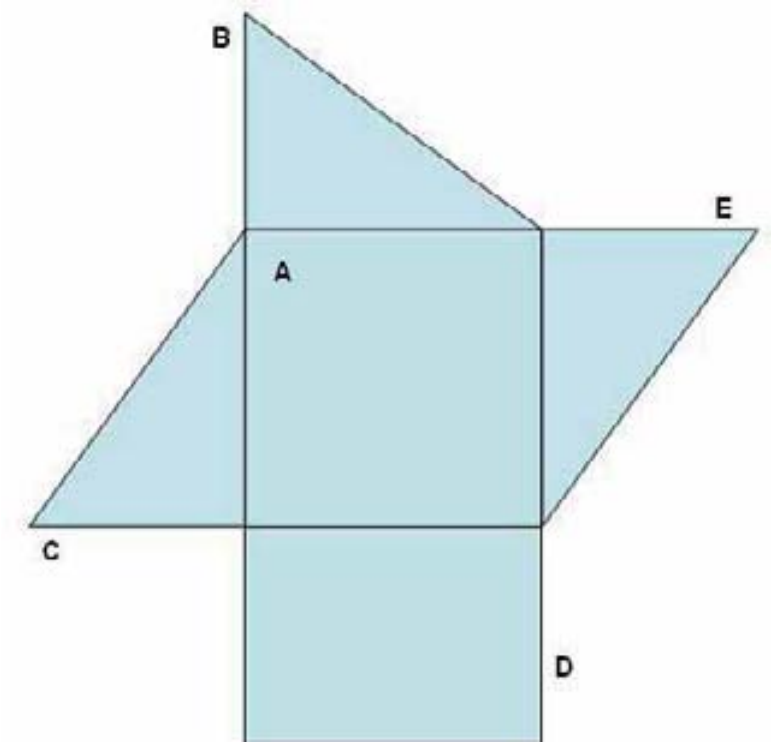
Elabore un programa que realice el cálculo de una regla de tres simple. Como sabe, dicha regla permite calcular la proporción de un número respecto a otro que actúa como referencia de la proporción. Este programa contesta preguntas del tipo ¿qué porcentaje de 200 es 135? El 200 sería el número base de la proporción, mientras que el 135 sería la proporción. La solución se calcula multiplicando la proporción por 100, y dividiéndolo entre el número base de la proporción.

Algoritmo orientado a datos

Miniespecificación

2.- Caso 7: Cálculo del área de un polígono compuesto

Observe la siguiente figura.



El programa debe calcular el área del polígono. La figura se compone de un cuadrado perfecto (A), tres triángulos rectángulos iguales (B, C, E), cuya base mide lo mismo que uno de los lados del cuadrado, y un rectángulo cuyo ancho mide lo mismo que uno de los lados del cuadrado, mientras que la altura mide lo mismo que la altura de los triángulos. Se pide que haga un programa que calcule el área total.

Por cultura general, sabemos que el área de un rectángulo se calcula multiplicando su base por su altura; el área de un triángulo rectángulo se calcula multiplicando su base por su altura, y luego dividiendo el resultado entre dos; finalmente, el área de un cuadrado se calcula elevando al cuadrado la medida de uno de sus lados.

La parte interesante de este problema es determinar el mínimo número de datos que hay que preguntar.

Algoritmo orientado a datos

Miniespecificación

3.- Caso 8: Determinando un descuento

Una tienda departamental ofrece descuentos directos sobre las compras que los clientes realizan.

El descuento normal va en función al tipo de cliente que la persona sea: A los clientes normales se les ofrece un 10 de descuento, y a los clientes premier un 15 de descuento.

Actualmente se tiene una promoción: en la compra de 5 artículos, se ofrecerá un descuento adicional al normal. Si el monto de la compra no excede de 1,000 pesos, se agrega un 5 de descuento; si el monto de la compra está entre 1,000 y 2,000 pesos, se agrega un descuento del 10 ; si la compra es mayor a 2,000 pesos, entonces el descuento adicional será del 15 .

En ese sentido, el menor descuento que se puede obtener es cuando un cliente normal compra menos de 1,000 pesos (15), y el máximo descuento que se puede obtener es cuando un cliente premier compra más de 2,000 pesos (30).

El programa deberá pedir la información de los precios de 5 productos, y determinar el monto de descuento aplicable.

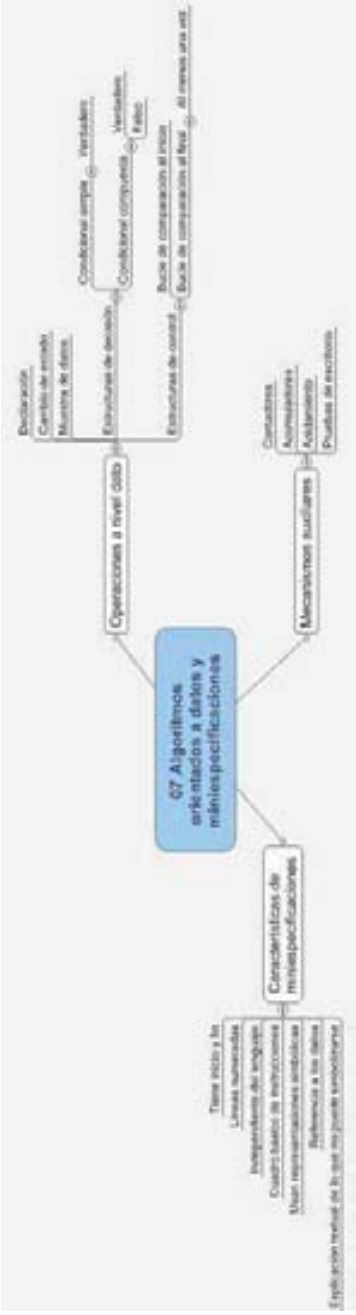
Utilice arreglos.

Algoritmo orientado a datos

Miniespecificación

FIN DEL EJERCICIO ✱

Mapa mental del capítulo



Terminología

« , 143
» , 141
Acumuladores, 146
Algoritmos orientados a datos, 138
Anidamiento, 149
Arreglos, 150
Bucle de comparación al final, 148
Bucle de comparación al inicio, 146
Bucle, 146
Código controlado, 143
Condicional compuesta, 144
Condicional simple, 144
Condicional, 144
Contadores, 145
Delimitador de fin, 143
Delimitador de inicio, 143
Delimitadores de la estructura, 143
Declaración, 140
Elemento, 150
Estado, 140
Estructura de control, 146
Estructura de decisión, 144
Finaliza una estructura, 143
Iniciando una estructura, 143
Item, 150
Loop, 146
Miniespecificación, 152
Muestra de datos, 142
Operación de asignación, 140
Operaciones a nivel dato, 139
Petición de datos, 141
Pruebas de escritorio, 163
Sangría, 144
Símbolo de muestra de datos, 143
Símbololo de petición, 141
Subíndice, 150

Preguntas

7.1 ¿Qué ventajas encuentra en las representaciones simbólicas de los hechos reales, para efectos del desarrollo de programas?

7.2 Dado que las miniespecificaciones son más completas y detalladas que los algoritmos, ¿Es usted partidario de eliminar los algoritmos y quedarnos sólo con miniespecificaciones como herrmamienta de abstracción de casos reales?

7.3 Elabore un análisis comparativo entre:

.3.1 Algoritmos.

.3.2 Algoritmos orientados a datos.

.3.3 Miniespecificaciones.

7.4 Continuando con el punto anterior, identifique al menos una característica valiosa de cada una de las herramientas, que no tengan las otras dos.

Examen rápido

Parte 1 : Marque co n una cruz el inciso que considere que responde a la pregunta.

1) Documentan las actividades secuenciales de un proceso, y los datos involucrados en las actividades.
(a) Miniespecificación
(b) Algoritmos orientados a datos
(c) Algoritmos

2) Operaciones que tienen como objetivo la declaración, cambio de estado y muestra de las variables.
(a) Operaciones primarias
(b) Miniespecificaciones
(c) Operaciones a nivel dato

3) Operación a través de la cual se cambia un estado a una variable.
(a) Asignación
(b) Declaración
(c) Instanciación

4) Es el símbolo de petición
(a) »
(b) «

5) Es la operación de salida por excelencia en un sistema.
(a) Petición de datos
(b) Informe
(c) Muestra de datos

Parte 2: Coloque en la línea la letra «F» si la sentencia es falsa, y «V» si la sentencia es verdadera.

6) Una variable puede existir, aún sin haber sido declarada.

7) En una condicional simple se puede especificar la acción por verdadero y por falso.

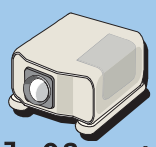
8) I ← 1 es un típico contador.

9) I ← I Valor es un típico acumulador.

10) Un bucle de comparación al inicio ejecuta el proceso al menos una vez.

8

Diagramas de flujo



lp08.ppt

Objetivos: **Aprenderá a desarrollar diagramas de flujo respetando los estándares de ANSI/ISO 5807-1985**

1. **Aprenderá en qué consiste un diagrama de flujo y cuáles son sus ventajas.**
2. **Aprenderá la importancia de los estándares en los símbolos de diagramas de flujo.**
3. **Aprenderá qué tipos de diagramas de flujo existen de acuerdo al estándar ANSI/ISO 5897-1985.**
4. **Aprenderá los símbolo s específicos para el de sarrollo de di agramas de flujo de programas, de acuerdo al estándar ANSI/ISO 5897-1985.**
5. **Aprenderá a desarrollar diagramas de flujo.**

www.detodoprogramacion.com

Contenido

Diagramas de flujo 177
Ventajas de utilizar diagramas de flujo 177
Estándar ANSI/ISO 5807-1985 para diagramas de flujo 178
Símbolos utilizados en los diagramas de flujo 178
Reglas para la elaboración de diagramas de flujo 181
Cómo elaborar un diagrama de flujo de programa 182
Diagramas de flujo de los casos prácticos 187
Caso 1: Mensaje 188
Caso 2: Niveles de servicio 189
Caso 3: Muestra de una tabla de multiplicar 191
Caso 4: Muestra las tablas de multiplicar del 1 al 5 193
Caso 5: Análisis de promedios 195
Comprobando habilidades en el diseño de diagramas de flujo 198
Mapa mental del capítulo 204
Terminología 205
Preguntas 205
Examen rápido 207

na imagen dice m s ue mil palabras.

Desde siempre, los símbolos han sido una herramienta de comunicación para los seres humanos; las representaciones gráficas ofrecen menos ambig edad que las expresiones textuales, en el sentido que son más concretos y mejor entendidos.

En las ciencias computacionales, los diagramas son utilizados ampliamente para clarificar la solución de problemas de procesamiento, esquematizar modelos y señalar términos de referencia.

Entendemos por *diagrama* al conjunto de símbolos que tienen un significado preestablecido, breves textos explicativos, y líneas que conectan a los elementos entre sí, relacionándolos y dándoles una secuencia.

Una parte indispensable de todo sistema de diagramación es que todas las personas interpreten los símbolos de la misma manera. Por esta razón, en este capítulo trataremos a los diagramas de flujo desde una perspectiva estandarizada.

www.detodoprogramacion.com

NOTA

Se tienen dos alternativas al diagramar: crear nuestra propia simbología, o utilizar una simbología estándar. Crear nuestra simbología tiene la máxima flexibilidad, pero nadie podrá interpretar nuestros diagramas a menos de que le expliquemos qué debe entender por cada símbolo. Una simbología estandarizada tiene la ventaja de que cualquier profesional que vea nuestros diagramas podrá interpretarlos. Otra ventaja es que el software para la elaboración de diagramas seguramente seguirá patrones estandarizados. Nosotros recomendamos para la elaboración de diagramas a Microsoft Visio y a RAPTOR.

Diagramas de flujo

Un *diagrama de flujo* es una representación visual del tratamiento de un problema; como mínimo, un diagrama de flujo muestra las operaciones realizadas dentro del proceso que pretende la solución del problema y la secuencia en la que son realizadas dichas operaciones. A grandes rasgos, un diagrama de flujo debe responder a las siguientes preguntas: ¿Qué actividades debo hacer para solucionar el problema? ¿En qué orden deberán ejecutarse dichas actividades?

Los diagramas de flujo generalmente son elaborados en las fases iniciales del ciclo de vida de desarrollo (análisis, diseño), dado que facilitan la comunicación entre los desarrolladores y los clientes; una de sus mayores virtudes es que facilitan el entendimiento de la lógica del problema a resolver, transformando problemas grandes y complejos en una serie ordenada de actividades por realizar.

Ventajas de utilizar diagramas de flujo

Las ventajas de utilizar diagramas de flujo son las siguientes.

1. Son generalmente conocidos e interpretados.
2. Son una de las mejores formas de comunicar la lógica de un programa o sistema.
3. Permiten el análisis de un problema de una forma abstracta.
4. Son aceptados como documentación técnica para los programas y desarrollos.
. Sirven como referencia para los programadores.
6. Apoyan el proceso de depuración de programas, sobre todo en la solución de problemas relacionados con la lógica del programa, mismos que el compilador es incapaz de detectar.
. Facilitan las funciones de mantenimiento a los programas, al exponerlos de una forma clara y lógica.

Estándar ANSI/ISO 5807-1985 para diagramas de flujo

De acuerdo al estándar ANSI ISO 5807 1985 de la (*merican ational tandard nstitute*), los diferentes tipos de diagramas de flujo son los siguientes.

1. Diagramas de flujo de datos (*data flo chart*).
2. Diagramas de flujo de programas (*program flo chart*).
3. Diagramas de flujo de sistemas (*system flo chart*).
4. Diagrama de flujo de red (*program net or chart*).
. Diagramas de recursos de sistemas (*system resources chart*).

De acuerdo al estándar, un *diagrama de flujo* es una representación gráfica de la definición, análisis y métodos de solución de problemas, en el cual los símbolos son utilizados para representar operaciones, datos, flujos, equipos, etcétera.

Un *s mbolo* es una representación gráfica asociada a un concepto de funcionalidad determinado. Dicha asociación, para ser efectiva, debe ser generalmente entendida y aceptada.

El estándar ANSI ISO 5807 1985 propone símbolos para todos los tipos de diagramas, pero hace la distinción respecto a los símbolos que aplican para cada tipo de diagrama en particular. Desde luego, nuestro interés se centra en los diagramas de flujo de programas.

Un *diagrama de flujo de programa* consiste en los siguientes elementos.

1. Símbolos de procesamiento que definen la ruta que seguirá el programa tomando en cuenta condiciones lógicas.
2. Líneas de flujo que indican el flujo de control de las operaciones.
3. Símbolos especiales que facilitan la lectura y escritura del diagrama de flujo.

Símbolos utilizados en los diagramas de flujo

De acuerdo al estándar ANSI ISO 5807 1985 los símbolos se dividen en las siguientes categorías:

- Símbolos de datos (*data symbols*).
- Símbolos de proceso (*process symbols*).
- Símbolos de línea (*line symbols*).
- Símbolos especiales (*special symbols*).

Los símbolos que aplican a los diagramas de flujo de los programas son los siguientes.

Símbolo	Categoría	Nombre y utilidad
	De datos	**Símbolo Datos** ***(Data)*** Representa a datos de entrada y salida. **Operaciones que soporta:** ❒ Petición de datos. ❒ Muestra de datos.
	De proceso	**Símbolo Proceso** ***(Process)*** Representa cualquier operación de procesamiento, por ejemplo, la ejecución de una o más operaciones que resultan en un cambio de valor, forma o ubicación de la información; también puede tratarse de operaciones para determinar una o más rutas que puede seguir el flujo del programa (sin que sea una decisión). **Operaciones que soporta:** ❒ Declaraciones. ❒ Cambios de estado.
	De proceso	**Símbolo Proceso predefinido** ***(Predefined Process)*** Representa un proceso nominado que consiste en una o más operaciones especificadas en otro lado, como podría ser una subrutina, una clase o un módulo externo. Puede ser utilizado para evitar la mezcla de operaciones de diferente nivel. **Operaciones que soporta:** ❒ Llamado a procedimientos.
	De proceso	**Símbolo Preparación** ***(Preparation)*** Este símbolo representa una modificación o instrucción que afecta la actividad subsecuente, aunque no forma parte del problema. Ejemplos de esto sería establecer una variable de trabajo o el inicializar contadores y acumuladores. **Operaciones que soporta:** ❒ Cambios de estado.
	De proceso	**Símbolo Decisión** ***(Decision)*** Representa una operación de decisión, que aunque tiene una sola entrada, puede tener varias salidas. Una y sólo una de las salidas será utilizada, y la selección de la salida está en función de la evaluación de una condición lógica. Cada salida se representa como líneas de flujo adyacentes, que representan las rutas que el proceso puede seguir. **Operaciones que soporta:** ❒ Estructuras de decisión.

Símbolo	Categoría	Nombre y utilidad
	De proceso	**Símbolo Modo paralelo** ***(Parallel mode)*** Representa la sincronización de dos o más operaciones que deben realizarse de forma dependiente (paralelo). El uso del modo paralelo hace obligatorio que para realizar un proceso sea necesaria la conclusión de otro proceso. **Operación que soporta:** ❒ Cambios de estado.
	De proceso	**Símbolo Límites de bucle** ***(Loop limit)*** Representa el inicio y fin de un bucle. La decisión a evaluar se encontrará en el límite inicial o en el final, dependiendo del tipo de ciclo que se esté utilizando. **Operaciones que soporta:** ❒ Estructuras de control.
	De línea	**Símbolo Línea de flujo** ***(Line)*** Representa el flujo de datos y el control de la operación del programa. Representa la secuencia lógica en que se desarrollan las operaciones.
	De línea	**Símbolo Línea punteada** ***(Dashed line)*** Representa una relación alternativa entre dos o más símbolos. Es útil para representar una conexión poco común entre dos o más elementos.
	Especiales	**Símbolo Conector** ***(Connector)*** Traslada el control del programa a otra parte dentro del mismo diagrama. Cada conector deberá contar con un identificador único.
	Especiales	**Símbolo Terminador** ***(Terminator)*** Representa la salida o la entrada al diagrama. Al menos se debe tener una entrada al diagrama (inicio), y al menos una terminación (fin).
	Especiales	**Símbolo Anotación** ***(Annotation)*** Agrega explicaciones que clarifiquen de forma descriptiva el funcionamiento del diagrama y sus elementos.

Reglas para la elaboración de diagramas de flujo

Las reglas básicas para la elaboración de diagramas de flujo son las siguientes.

1. Deben utilizarse símbolos estándar para la diagramación, aplicables a los diagramas de flujo de programas.
2. Todos los requerimientos y actividades del programa deberán representarse en orden lógico.
3. El diagrama de flujo debe ser claro, limpio y fácil de seguir.
4. Debe evitarse cualquier ambig edad en el entendimiento del diagrama de flujo. Si el diagrama no simplifica el entendimiento del proceso diagramado, hay que redefinirlo.

. La dirección usual en que debe dibujarse un diagrama de flujo es de izquierda a derecha y de arriba a abajo (*left right*, *top do n*).

6. Las **líneas de flujo** deben correr de forma vertical y horizontal, en caso de no ser posible mantenerlas rectas, se deben utilizar ángulos rectos. No deben trazarse en diagonal.

. Las **líneas de flujo** pueden terminar en punta de flecha, para sugerir la dirección del flujo de control del programa.

. Sólo una **línea de flujo** debe salir de un símbolo de **proceso**.

. Sólo un flujo de datos debe entrar a un símbolo **decisión**, pero es posible que salgan de él dos o tres líneas de flujo, dependiendo las posibles salidas permitidas por el proceso.

10. Sólo una línea de flujo puede estar asociada a un símbolo **terminador**.
11. En los símbolos debe evitarse al máximo la información textual. La breve información textual asociada a un elemento debe estar dentro de sus límites gráficos.
12. En la información textual de los símbolos, se debe iniciar preferentemente con un verbo en infinitivo. Correcto: “Imprimir documento”. Incorrecto: “El documento pasa a impresión”, o “Impresión del documento”.
13. En caso de que se requieran explicaciones que no sean breves, deberá utilizarse el símbolo **anotaciones**.
14. En el caso de los símbolos **datos**, que permiten especificar entradas y salidas, se recomienda agregar en el contenido textual las palabras GET y PUT, dependiendo si se trata de acciones de petición o de mostrado de datos, respectivamente.

1 . Si el diagrama se hace complejo, es mejor utilizar símbolos conectores para reducir el número y amplitud de las líneas de flujo.

16. Si el diagrama se hace muy extenso, es preferible crear niveles de abstracción distintos, de tal forma que en el primer nivel el diagrama quede claro; a partir de ahí podrán hacerse diagramas para los elementos que puedan ser descritos a mayor detalle.

www.detodoprogramacion.com

1 . Hay que asegurar que el diagrama tiene un **terminador** de inicio y un **terminador** de fin.

1 . Se recomienda hacer una prueba de escritorio para probar que el diagrama funciona adecuadamente.

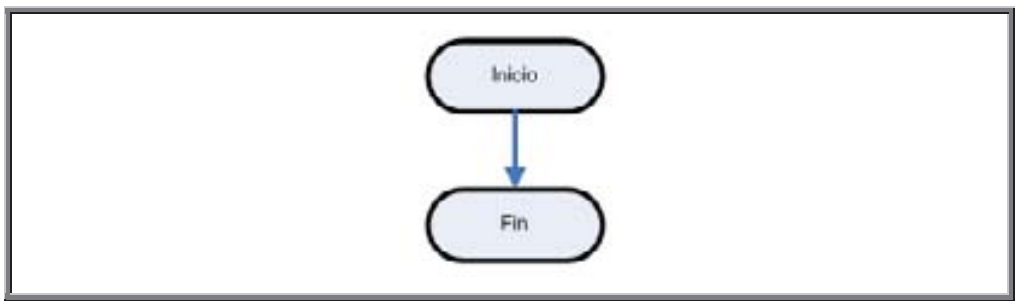
Cómo elaborar un diagrama de flujo de programa

Ejemplificaremos la creación de un diagrama de flujo de un programa con un programa que estará preguntando números del 1 al 10. Sólo terminará su ejecución al momento en que proporcione un número 7.

El algoritmo sería como sigue.

1. Inicio.
2. Se pregunta un número del 1 al 10 (Número,i[1-10]).
3. Si el número es igual a 7, entonces el programa concluye.
4. Si el número no es 7, entonces el programa manda un mensaje que diga que el número no es el esperado.
5. Fin.

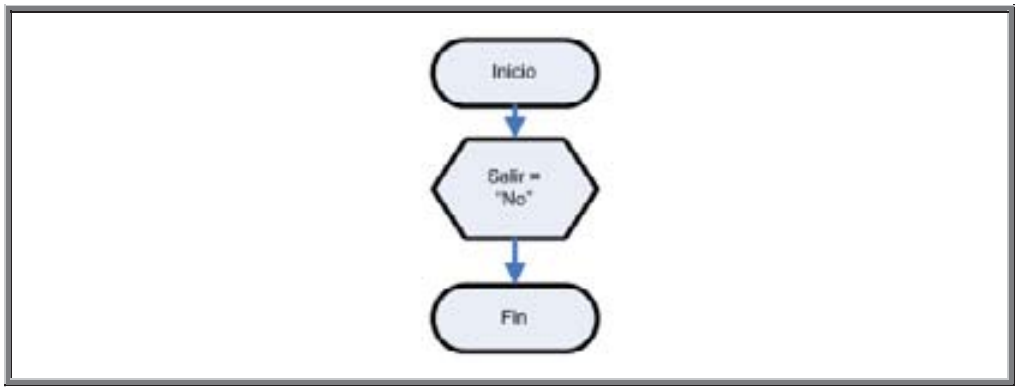
Inicio y fin. Como todos los diagramas deben tener principio y fin, el primer paso es colocar los símbolos de inicio y fin. Para ello se utiliza el símbolo **terminador**.



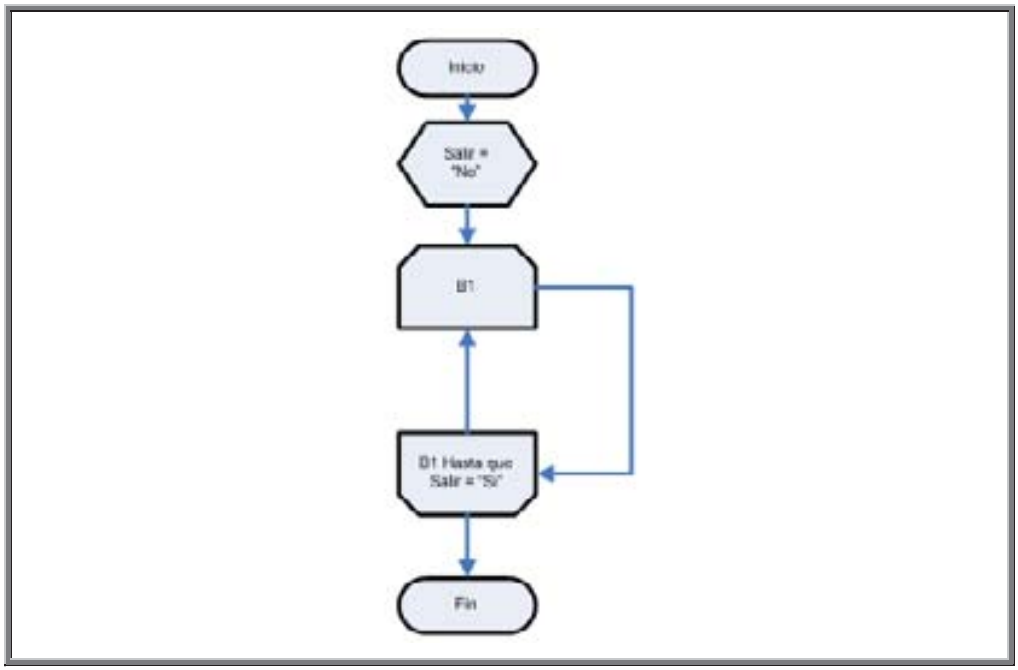
Preparación. Como es un ciclo que estará repitiéndose un número indeterminado de veces, agregamos una variable de control de proceso (Salida,x(2)). De inicio valdrá "No"; cuando el número sea 7, cambiará su valor a "Si", con lo que provocará la terminación del programa.

Vea cómo se especifica con el símbolo de **preparación**, dado que es una variable de trabajo que no forma parte de la lógica original del programa. Esto es común cuando se utilizan contadores, acumuladores y variables de tipo bandera (*flag*), que tienen como misión controlar la ejecución del código del programa.

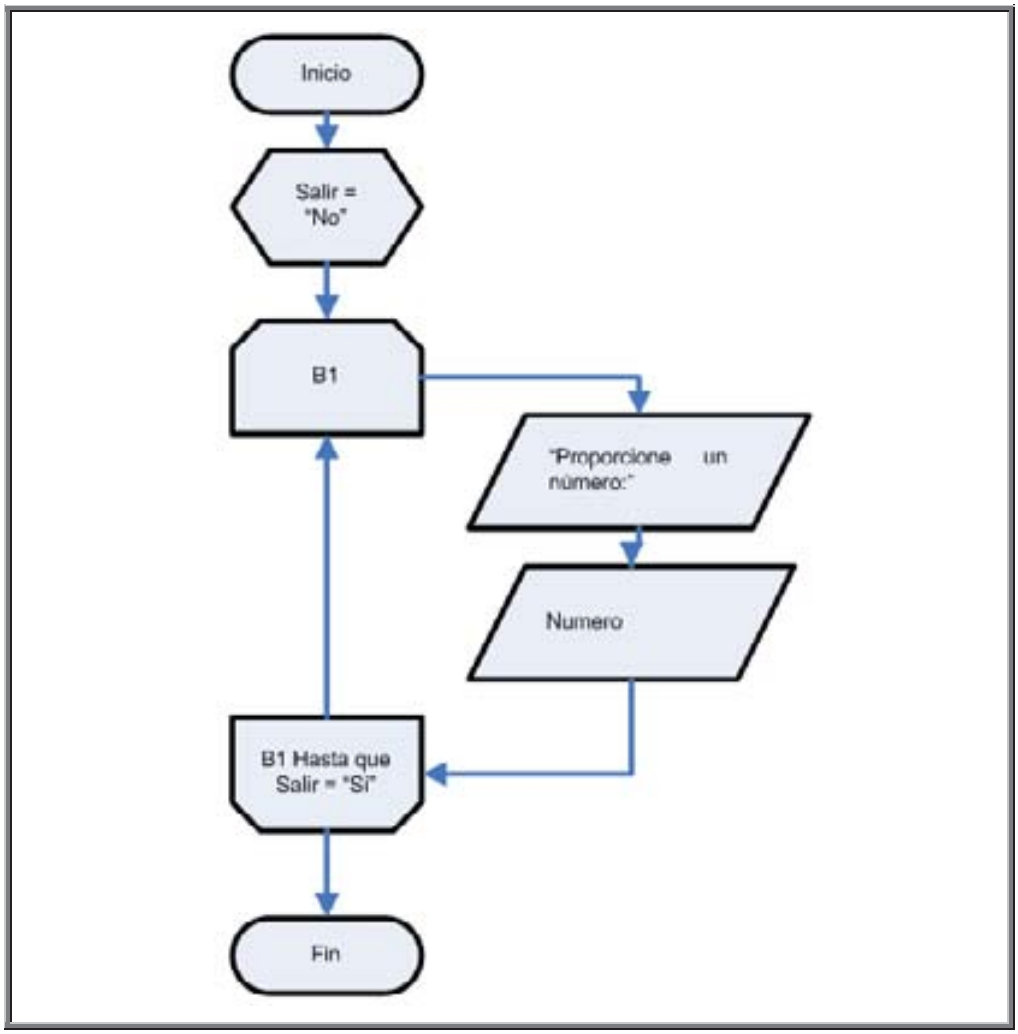
Vea cómo las líneas de flujo aparecen con punta de flecha. Esto no es requerido por el estándar ANSI, pero se permite para hacer énfasis de la dirección del flujo del control del programa; este aspecto es útil cuando el diagrama está saturado y la lectura de izquierda derecha, arriba abajo es complicada de mantener.



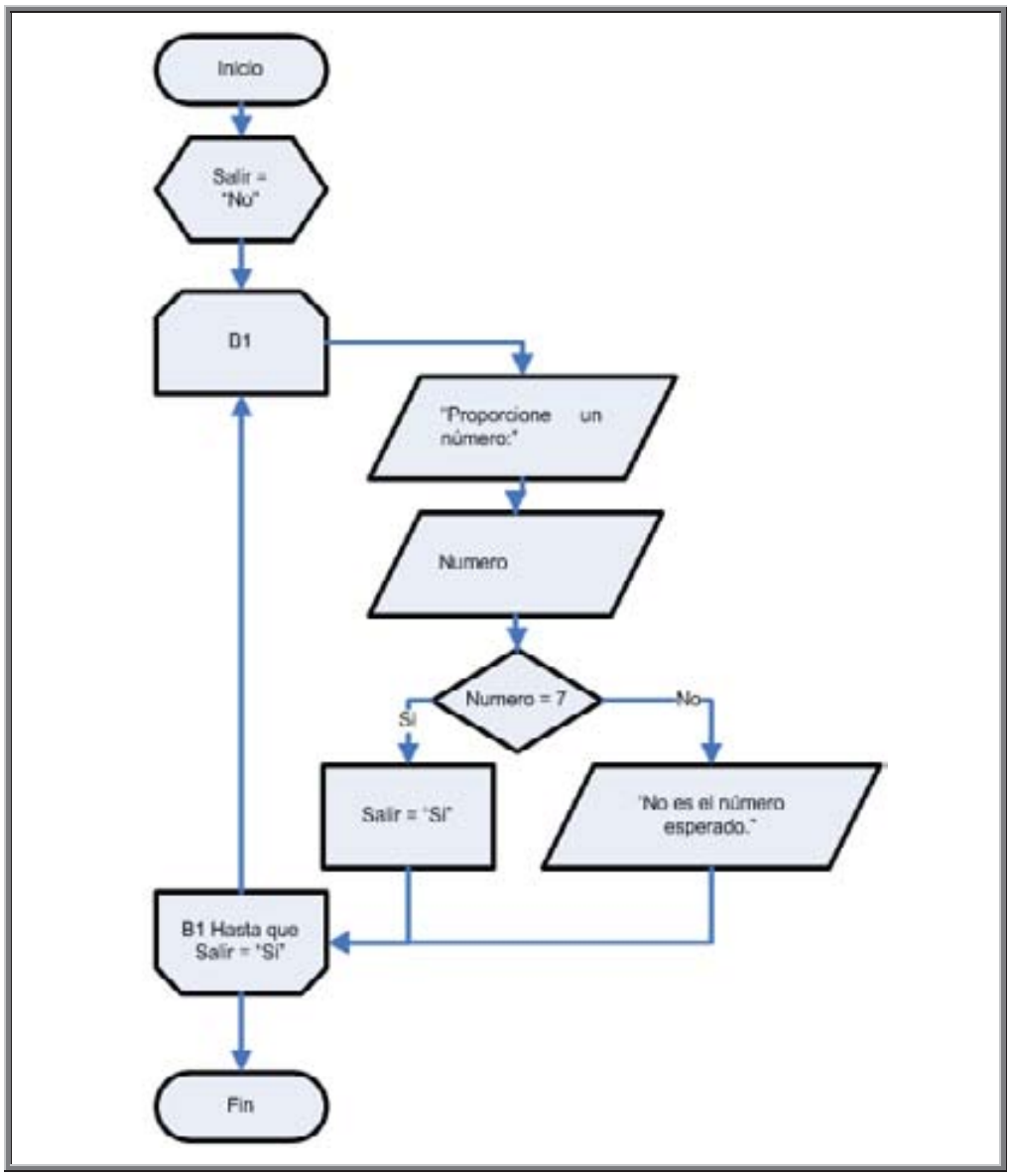
Bucles. Dado que se estará preguntando de forma reiterada un número, se integra un bucle al diagrama (B1).



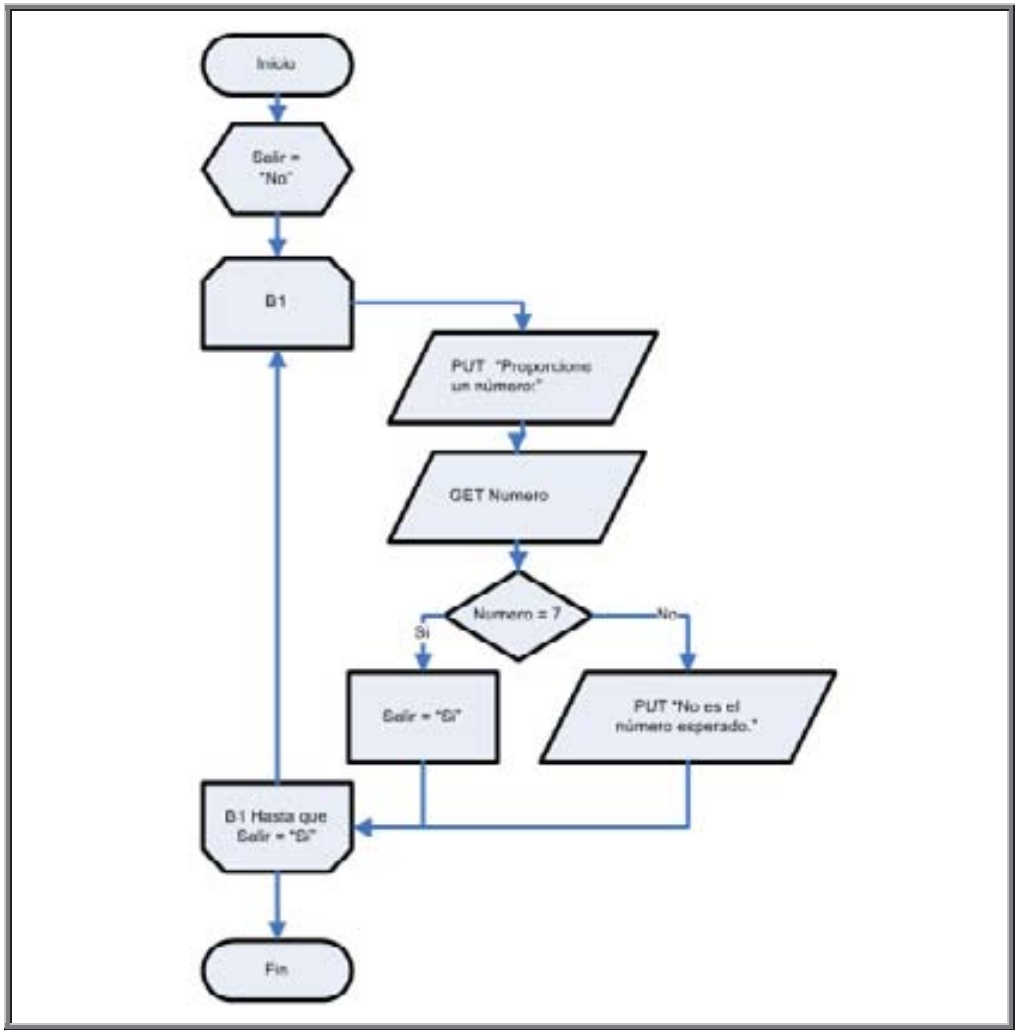
Pidiendo y mostrando datos. Para solicitar y mostrar datos se utiliza el símbolo **dato**. En nuestro ejemplo, uno muestra el texto que solicita el dato (mostrado de datos), mientras que el otro pregunta el dato (petición de datos), que se asignará a la variable Numero.



Manejo de condicionales. Las condicionales se manejan a través de símbolos **decisión**. Se evalúa si el número es 7. Si el número es 7, se asigna el valor de "Si" a la variable Salir, con lo que la condición del bucle se cumple, dando por concluido el programa. En caso de que el número no sea 7, Salir seguirá manteniendo su valor de "No", la condición del bucle no se cumplirá, y todo el proceso se ejecuta nuevamente.



Uso de GET y PUT. Como podrá darse cuenta, los símbolos **Datos** son iguales para la petición de datos que para la muestra de datos. Hacemos algunas modificaciones en el texto del símbolo, a fin de darles un sentido específico. Si el símbolo hace referencia a una petición de datos, antepondremos al nombre de la variable la palabra GET (*conseguir*); si el símbolo hace referencia a una muestra de datos, antepondremos al mensaje de salida la palabra PUT (*colocar*). Nuestro diagrama quedaría como sigue.



El uso de GET y PUT no forma parte del estándar ANSI.

Diagramas de flujo de los casos prácticos

En esta parte se construirán los diagramas de flujo de los casos prácticos a partir de las miniespecificaciones. Cabe aclarar que no es necesario haber realizado las miniespecificaciones para realizar los diagramas de flujo; el diagrama de flujo puede hacerse directamente de la descripción del problema, a partir del algoritmo y del algoritmo orientado a datos.

Dependiendo de lo que tomemos en cuenta como base, será el nivel de detalle del diagrama. Nosotros tomaremos como origen las miniespecificaciones, a fin de obtener los diagramas al mayor detalle. Dado lo pequeño de los programas, sólo utilizaremos los siguientes símbolos.

Símbolo	Nombre y utilidad
	Datos ***(Data)*** **Operaciones que soporta:** ❒ Petición de datos. ❒ Muestra de datos.
	Proceso ***(Process)*** **Operaciones que soporta:** ❒ Declaraciones. ❒ Cambios de estado.
	Preparación ***(Preparation)*** **Operaciones que soporta:** ❒ Cambios de estado.
	Decisión ***(Decision)*** **Operaciones que soporta:** ❒ Estructuras de decisión.
	Límites de bucle ***(Loop limit)*** **Operaciones que soporta:** ❒ Estructuras de control.
	Línea de flujo ***(Line)***
	Terminador ***(Terminator)***

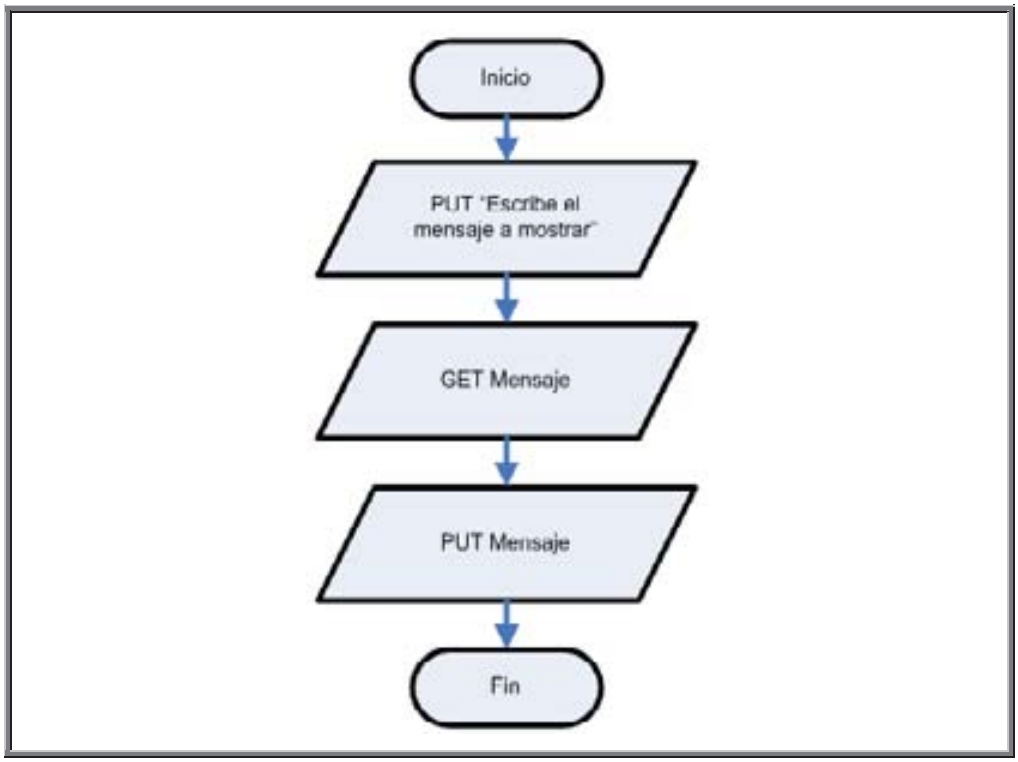
Caso 1: Mensaje

Este programa tiene como finalidad comprobar técnicas y herramientas de desarrollo. Lo que el programa debe hacer es preguntar por un mensaje de saludo, para posteriormente mostrar el mensaje que se le proporcionó.

Miniespecificación

```
1   INICIO:
2           (Mensaje, x(30))
3           « "Escribe un mensaje a mostrar: "
4           » Mensaje
5           « Mensaje
6   :FIN
```

Diagrama de flujo del programa



Caso 2: Niveles de servicio

Se desea saber la disponibilidad del servicio de Internet para una persona, expresado en horas de disponibilidad.

Dicho nivel de servicio depende del grado de disponibilidad que ofrece la computadora que utiliza, así como del grado de disponibilidad del enlace a Internet que utiliza la máquina.

El horario de labores de la persona que utiliza la computadora y el enlace a Internet es de 8 horas.

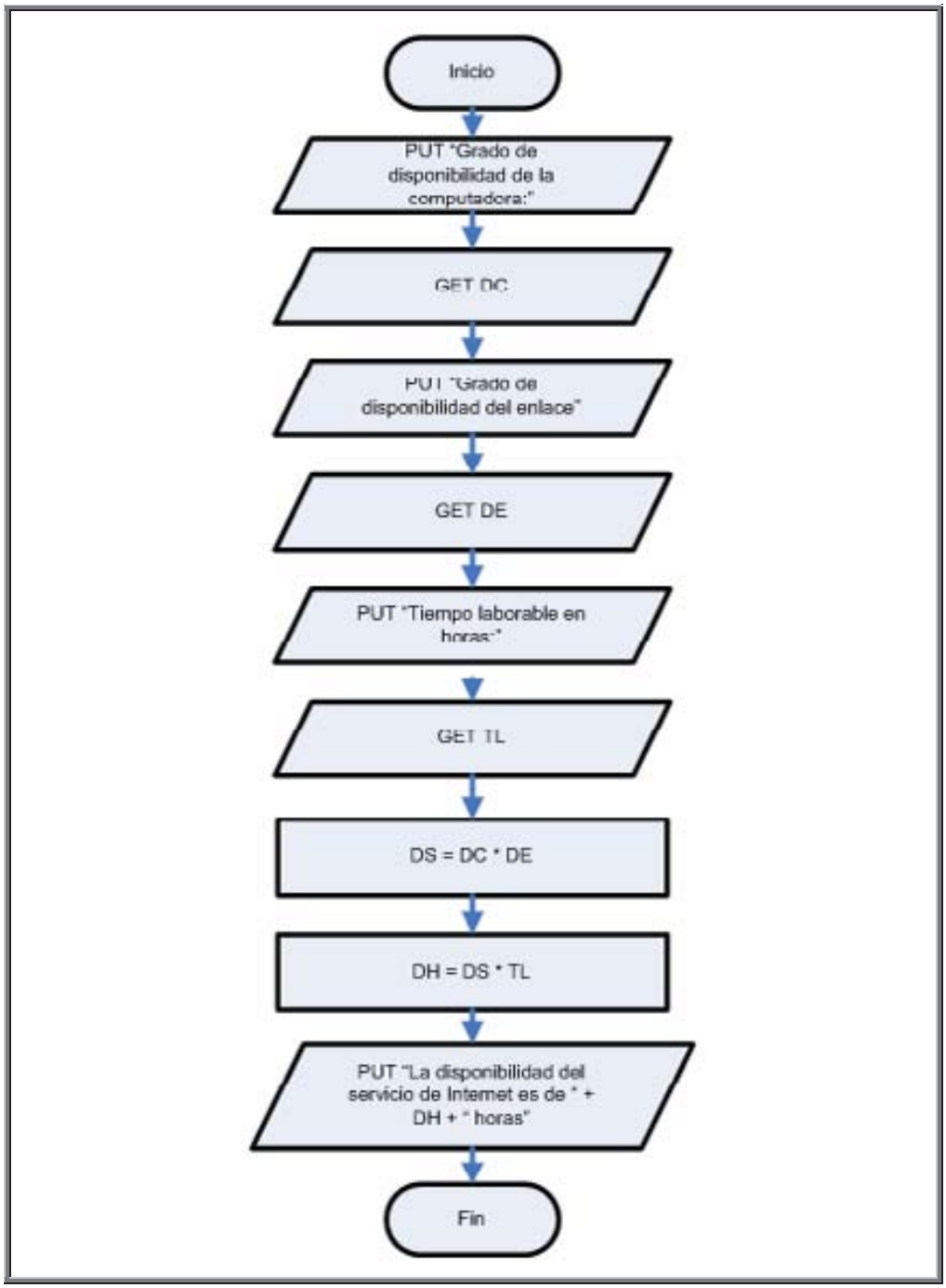
Para poder realizar los cálculos adecuadamente, tome en cuenta que la disponibilidad de un sistema de dispositivos (dos o más dispositivos que trabajan de forma conjunta) se calcula multiplicando los porcentajes de grado de disponibilidad de cada uno de ellos, y el resultado siempre será menor a la disponibilidad del menor.

Para calcular el tiempo de disponibilidad de un servicio (en este caso el acceso a la información publicada en Internet), sólo basta con multiplicar el tiempo en que es requerido el servicio por la disponibilidad del sistema de dispositivos.

Miniespecificación

```
1   INICIO:
2           (DC,d[0-1])
3           (DE,d[0-1])
4           (TL,d[0-1])
5           (DS,d[0-1])
6           (DH,d[0-1])
7           « "Grado de disponibilidad de la computadora:"
8           » DC
9           « "Grado de disponibilidad del enlace:"
10          » DE
11          « "Tiempo laborable en horas:"
12          » TL
13          DS ← DC * DE
14          DH ← DS * TL
15          « "La disponibilidad del servicio de Internet es de " + DH + " horas"
16  :FIN
```

Diagrama de flujo del programa



NOTA

TIP DEL PROGRAMADOR: EVITANDO UN BUCLE INFINITO

Un bucle con condición al final puede ser infinito por dos razones: porque la condición que se evalúa en el bucle nunca es verdadera (por ejemplo una condición lógica compuesta y disyuntiva, donde siempre una de las condiciones es False), o porque la variable de trabajo no está cambiando en dirección a la condición.

Las recomendaciones son evitar al máximo las expresiones compuestas en la condición del bucle, a menos de que las sometamos a una prueba de escritorio que nos diga su comportamiento.

Cuide los siguientes tres puntos: ***a)*** Asegúrese que antes del bucle se inicializa la variable de trabajo. ***b)*** Asegúrese que dentro del bucle la variable de trabajo tiene incremento o decremento dentro del bucle, y ***c)*** cuide que el incremento o decremento haga que la condición siempre esté más próxima a cumplirse en el caso de Until, y a dejar de cumplirse, en el caso de While, con cada iteración.

Caso 3: Muestra de una tabla de multiplicar

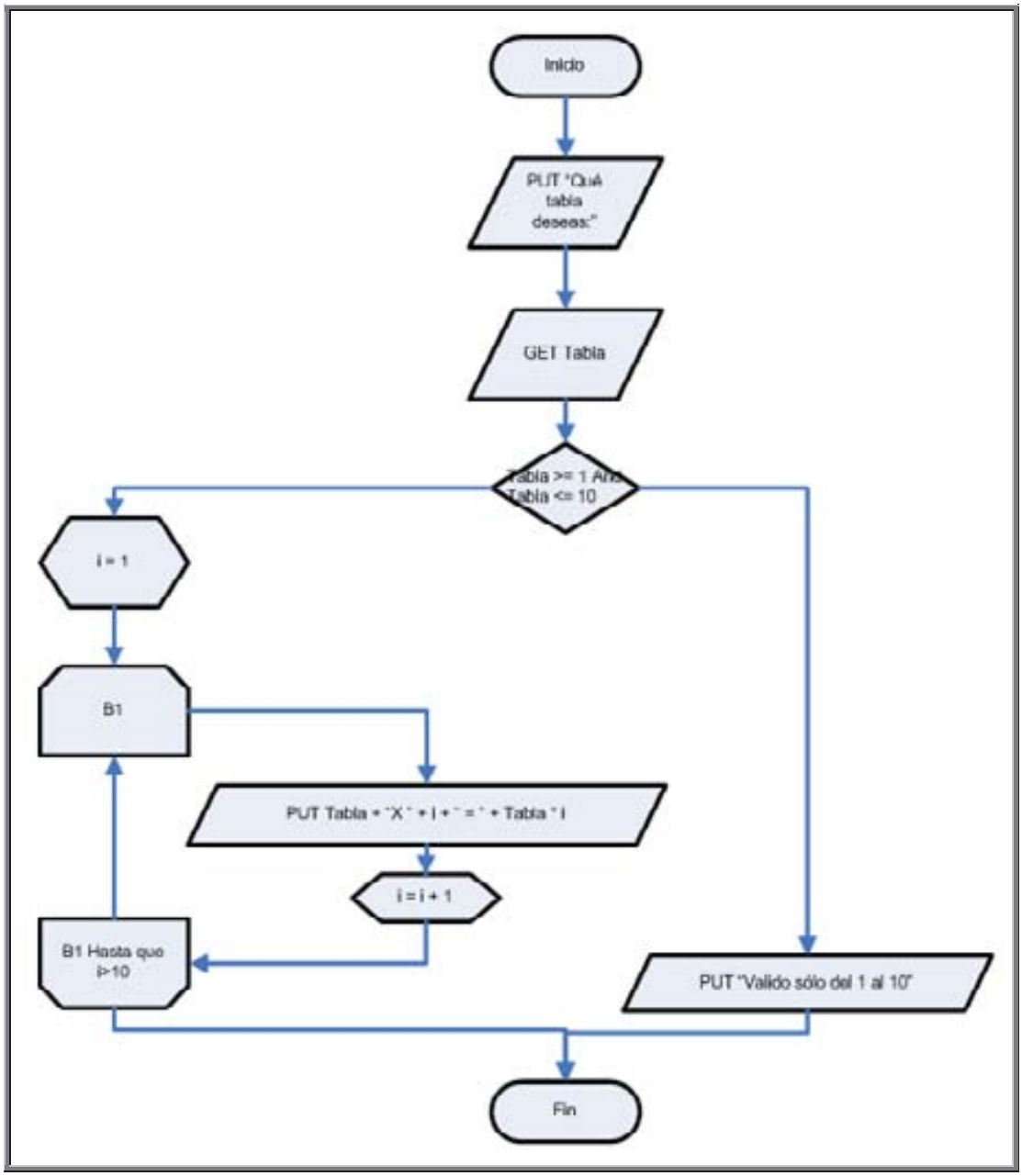
El programa debe mostrar cualquier tabla de multiplicar que le solicitemos. La única restricción es que la tabla deberá ser de un número mayor o igual a 1, y menor o igual a 10.

Si se pide una tabla de un número que no esté entre el 1 y el 10, el programa debe indicar, mediante un mensaje, que no puede continuar.

Miniespecificación (usando estructuras de control)

```
1   INICIO:
2           (i,i[0-n])
3           (Tabla,i[1-10])
4           « "Qué tabla deseas: "
5           » Tabla
6           Si Tabla >= 1 ∧ Tabla <=10 Entonces
7                   i ← 1
8                   Inicio
9                           « Tabla + " X" + i + "=" + Tabla * i
10                          i ← i + 1
11                  Hasta que i > 10
12          De lo contrario
13                  « "Válido sólo del 1 al 10"
14          Fin Si
    :FIN
```

Diagrama de flujo del programa



Caso 4: Muestra las tablas de multiplicar del 1 al 5

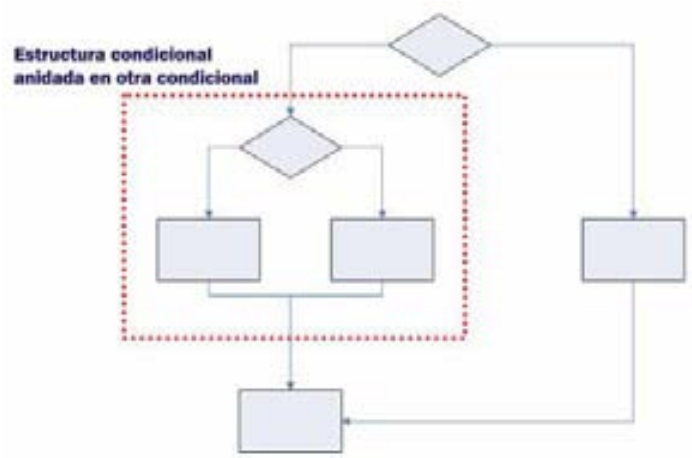
El programa debe mostrar todas las tablas de multiplicar del 1 al 5, incluyendo un encabezado por tabla.

La forma más práctica de hacer este programa es anidando dos estructuras de control.

La forma en que se anidan dos estructuras en un diagrama, es que una estructura se encuentre completamente dentro del flujo del proceso de otra estructura de mayor amplitud.

Figura 08.01

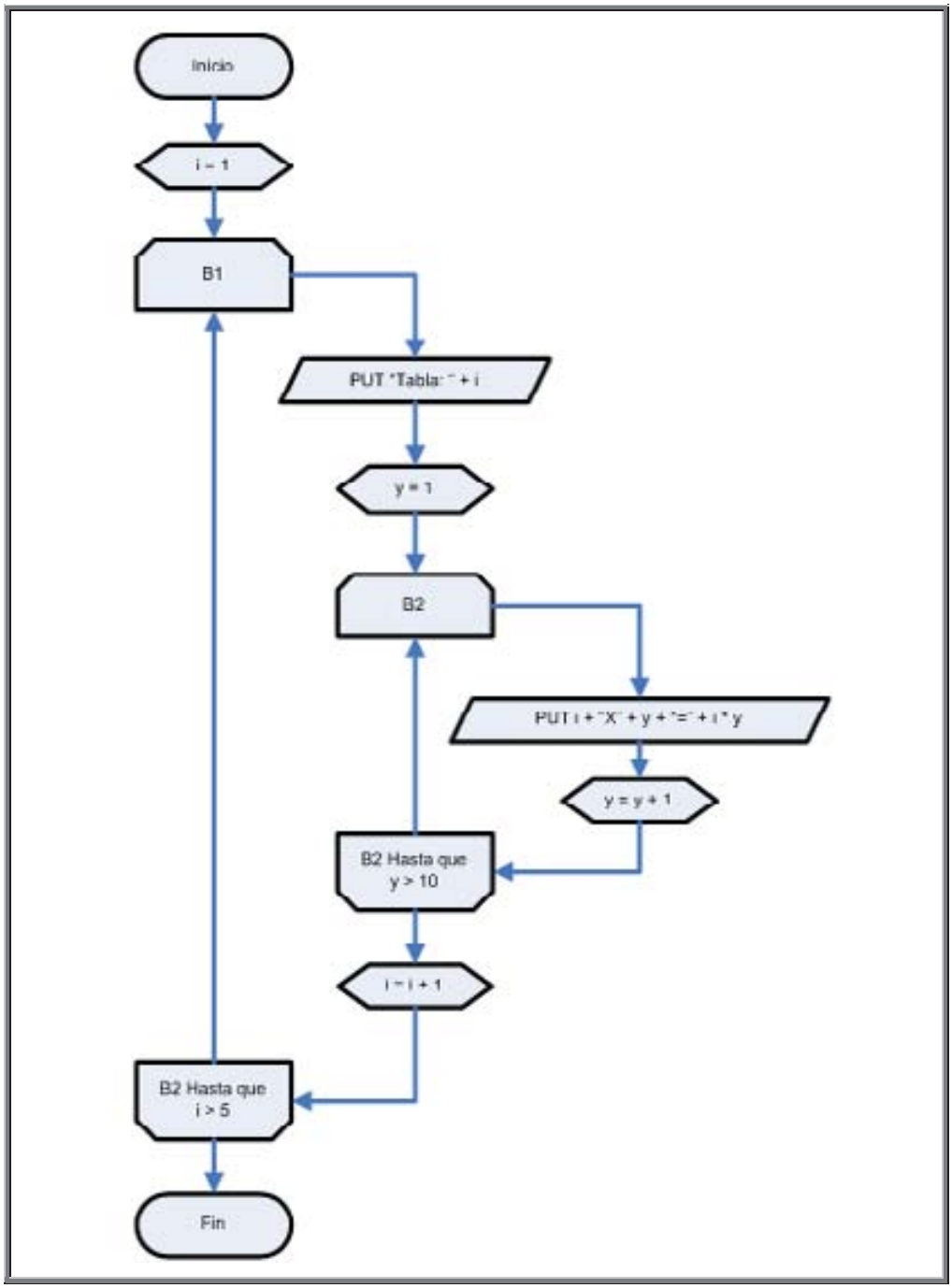
Estructuras anidadas



Miniespecificación

```
1   INICIO:
2           (i,i[0-n])
3           (y,i[0-n])
4           i ← 1
5           Inicio
6                   « "TABLA: " + i
7                   y ← 1
8                   Inicio
9                           « i + " X" + y + "=" + i * y
10                          y ← y + 1
11                  Hasta que y > 10
12                  i ← i + 1
13          Hasta que i > 5
14  :FIN
```

Diagrama de flujo del programa



Caso 5: Análisis de promedios

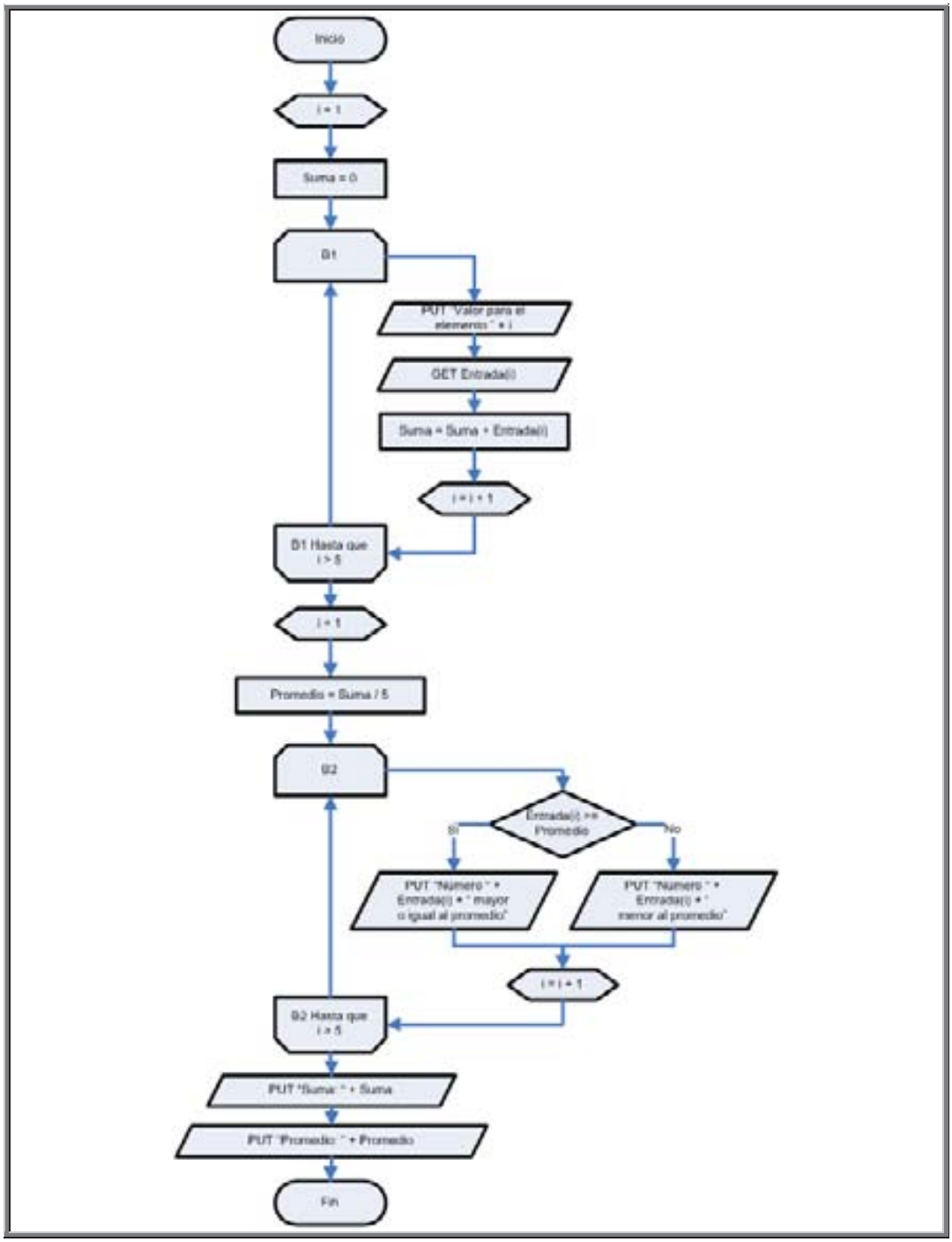
El programa debe pedir 5 números, para después mostrarlos indicando si están o no por encima de su promedio. Al final deberá enviarse un resumen en donde se indique la suma total de los números, y el promedio calculado.

Para la petición de datos utilice un arreglo.

Miniespecificación (usando estructuras)

```
1   INICIO:
2           (i,i[0-n])
3           (Entrada(5),i[0-n])
4           (Suma,d[0-n])
5           (Promedio,d[0-n])
6           Suma ← 0
7           i ← 1
8           Inicio
9                   « “Valor para el elemento ” + i + “: “
10                  » Entrada(i)
11                  Suma ← Suma + Entrada(i)
12                  i ← i + 1
13          Hasta que i > 5
14          Promedio ← Suma / 5
15          i ← 1
16          Inicio
17                  Si Entrada(i) >= Promedio Entonces
18                      « “Número ” + Entrada(i) + “ mayor o igual al promedio: “
19                  De lo contrario
20                      « “Número ” + Entrada(i) + “ menor al promedio: “
21                  Fin Si
22                  i ← i + 1
23          Hasta que i > 5
24          « “Suma: ” + Suma
25          « “Promedio: ” + Promedio
26  :FIN
```

Diagrama de flujo del programa



Recomendamos ampliamente utilizar una herramienta computarizada para la elaboración de diagramas de flujo, ya que de otra forma la tarea puede convertirse en una actividad tediosa, ante la imposibilidad de modificar un diagrama escrito sobre el papel. Puede utilizar soft are como Microsoft Visio y RAPTOR.

Figura 08.02

Interfaz de Microsoft Visio

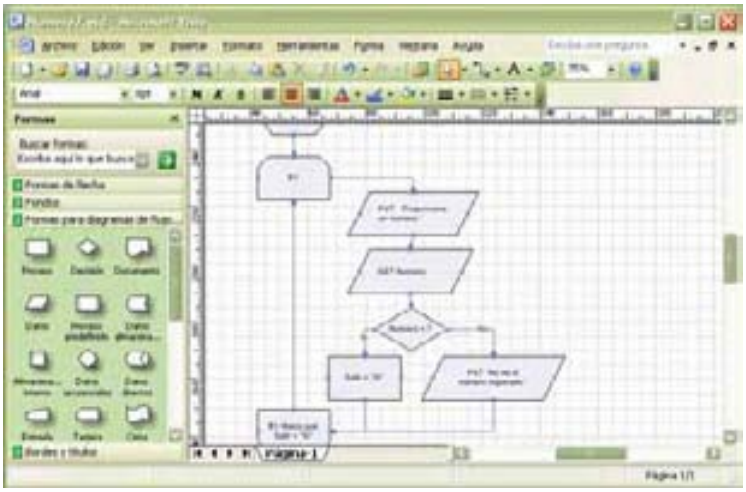
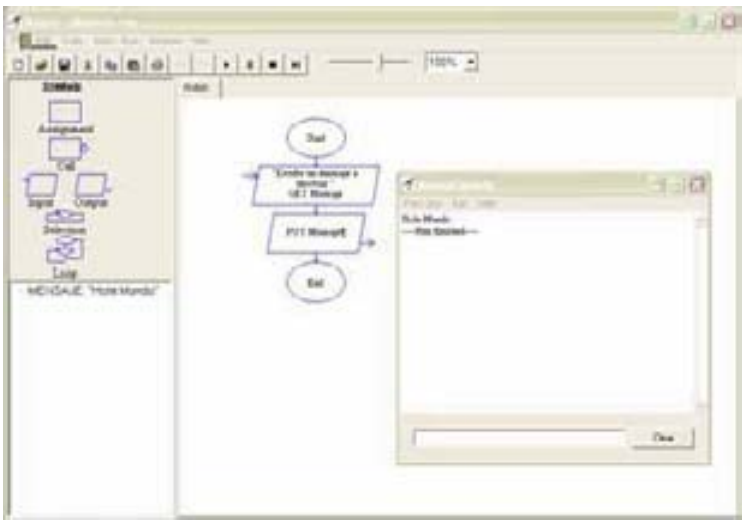


Figura 08.03

Interfaz de RAPTOR



NOTA

TIP DEL PROGRAMADOR: INTERCAMBIO DE VALORES

Es posible que se tope alguna vez con la necesidad de que una variable (var1) intercambie el valor con otra variable (var2), sin que ninguna de las dos pierda su valor. Esto se hace fácil con una variable de trabajo (Trabajo) del mismo tipo de dato que las anteriores. El proceso es el siguiente:

Trabajo ← Var1

Var1 ← Var2

Var2 ← Tabajo

Ejercicio 08.01

Comprobando habilidades en el diseño de diagramas de flujo

Elabore lo siguiente para cada uno de los casos que se propongan.

- ❒ Diagrama de flujo.

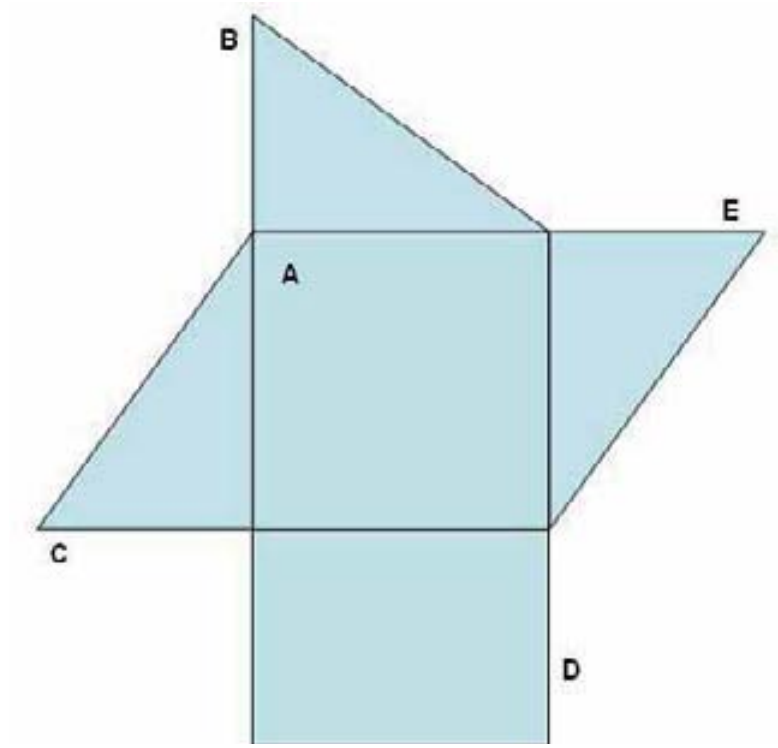
1.- Caso 6: Regla de tres simple

Elabore un programa que realice el cálculo de una regla de tres simple. Como sabe, dicha regla permite calcular la proporción de un número respecto a otro que actúa como referencia de la proporción. Este programa contesta preguntas del tipo ¿qué porcentaje de 200 es 135? El 200 sería el número base de la proporción, mientras que el 135 sería la proporción. La solución se calcula multiplicando la proporción por 100, y dividiéndolo entre el número base de la proporción.

Diagrama de flujo del programa

2.- Caso 7: Cálculo del área de un polígono compuesto

Observe la siguiente figura.



El programa debe calcular el área del polígono. La figura se compone de un cuadrado perfecto (A), tres triángulos rectángulos iguales (B, C, E), cuya base mide lo mismo que uno de los lados del cuadrado, y un rectángulo cuyo ancho mide lo mismo que uno de los lados del cuadrado, mientras que la altura mide lo mismo que la altura de los triángulos. Se pide que haga un programa que calcule el área total.

Por cultura general, sabemos que el área de un rectángulo se calcula multiplicando su base por su altura; el área de un triángulo rectángulo se calcula multiplicando su base por su altura, y luego dividiendo el resultado entre dos; finalmente, el área de un cuadrado se calcula elevando al cuadrado la medida de uno de sus lados.

La parte interesante de este problema es determinar el mínimo número de datos que hay que preguntar.

Diagrama de flujo del programa

NOTA

TIP DEL PROGRAMADOR: RESOLUCIÓN LÓGICA NUMÉRICA

Imagine que tiene una expresión lógica compuesta por tres condiciones (I1 ∧ I2 ∨ I3). Además de saber si la expresión resolvió por falso o por verdadero, necesita saber que combinación de valores provocaron el resultado. La solución es asignar un número primo a cada valor verdadero. Si I1 Entonces Código ← Código + 3, Si I2 Entonces Código ← Código + 7, Si I3 Entonces Código ← Código + 11. De esa forma cada combinación tendrá un ID único:

Ii	I2	I3	Expresión	Código
False	False	False	False	0
False	False	True (11)	False	11
False	True (7)	True (11)	True	18
True (3)	True (7)	True (11)	True	21
True (3)	False	False	False	3
True (3)	True (7)	False	False	10
False	True (7)	False	False	7
True (3)	False	True (11)	True	14

3.- Caso 8: Determinando un descuento

Una tienda departamental ofrece descuentos directos sobre las compras que los clientes realizan.

El descuento normal va en función al tipo de cliente que la persona sea: A los clientes normales se les ofrece un 10 de descuento, y a los clientes premier un 15 de descuento.

Actualmente se tiene una promoción: en la compra de 5 artículos, se ofrecerá un descuento adicional al normal. Si el monto de la compra no excede de 1,000 pesos, se agrega un 5 de descuento; si el monto de la compra está entre 1,000 y 2,000 pesos, se agrega un descuento del 10 ; si la compra es mayor a 2,000 pesos, entonces el descuento adicional será del 15 .

En ese sentido, el menor descuento que se puede obtener es cuando un cliente normal compra menos de 1,000 pesos (15), y el máximo descuento que se puede obtener es cuando un cliente premier compra más de 2,000 pesos (30).

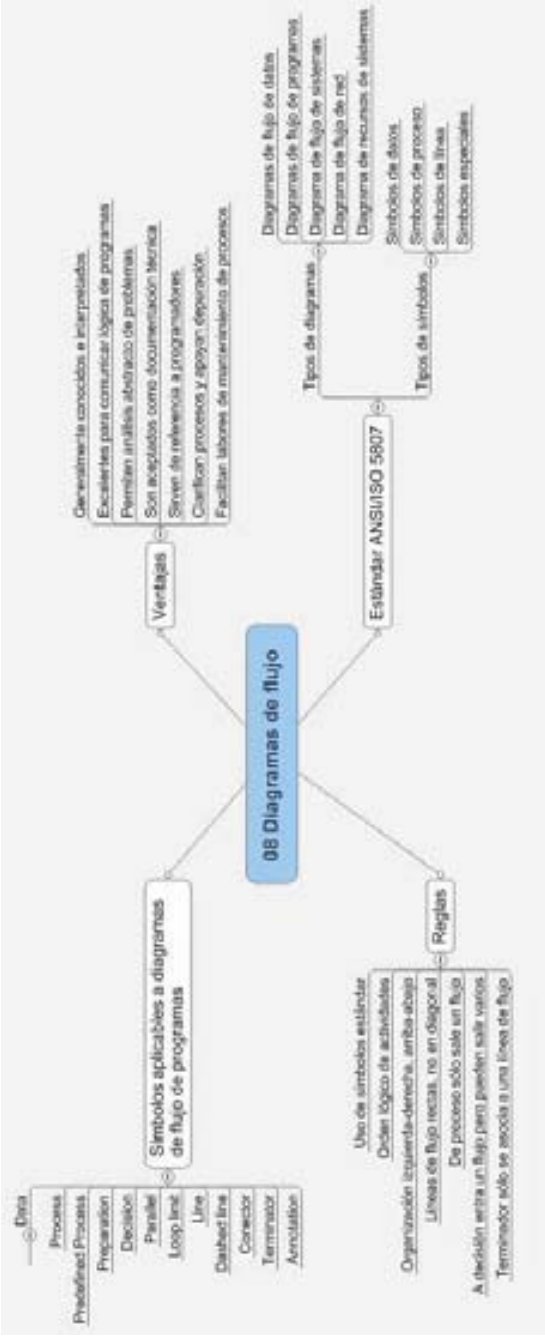
El programa deberá pedir la información de los precios de 5 productos, y determinar el monto de descuento aplicable.

Utilice arreglos.

Diagrama de flujo del programa

FIN DEL EJERCICIO ✱

Mapa mental del capítulo



Terminología

American National Standard Institute, 178
Annotation, 180
ANSI, 178
Connector, 180
Dashed line, 180
Data, 179
Decision, 179
Diagrama de flujo de programa, 178
Diagrama de flujo, 177
Diagrama de flujo, 178
Diagrama, 176
Line, 180
Loop limit, 180
Parallel mode, 180
Predefined Process, 179
Preparation, 179
Process, 179
Símbolo Anotación, 180
Símbolo Conector, 180
Símbolo Datos, 179
Símbolo Decisión, 179
Símbolo Límites de bucle, 180
Símbolo Línea de flujo, 180
Símbolo Línea punteada, 180
Símbolo Modo paralelo, 180
Símbolo Preparación, 179
Símbolo Proceso Predefinido, 179
Símbolo Proceso, 179
Símbolo Terminador, 180
Símbolo, 178
Terminator, 180

Preguntas

8.1 En su opinión, si ya cuenta con la miniespecificación del programa, ¿Qué objeto tendría elaborar el diagrama de flujo?

8.2 Además del estándar ANSI ISO 0 , ¿Qué otros estándares existen para la elaboración de diagramas de flujo?

8.3 ¿Qué consecuencias se pueden tener si se mezclan símbolos de diferentes tipos de diagrama en un solo diagrama?

8.4 Si usted se pone creativo y genera sus propios símbolos de diagrama, ¿Qué efecto tendrá en la forma en que sus diagramas son apreciados por la comunidad mundial de desarrolladores?

www.detodoprogramacion.com

Notas:

Examen rápido

Parte 1 : Marque co n una cruz el inciso que considere que responde a la pregunta.

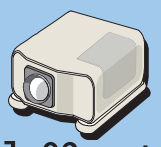
1) Es el estándar internacional más reconocido para la elaboración de diagramas de flujo.
(a) ANSI ISO 0 1
(b) ISO 000
(c) ANSI

race l neas conectan o los s bolos s tili a ea el e e lo

Símbolo		Nombre y utilidad
		Cambios de estado (variables de trabajo).
		Estructuras de control.
		Estructuras de decisión.
		Cambios de estado (variables de proceso).
		Declaraciones.
		Petición de datos.
		Muestra de datos.

9

Programación visual usando Raptor



lp09.ppt

Objetivos: Aprenderá a desarrollar programas utilizando Raptor.

1. Aprenderá qué es Raptor y cuáles son sus capacidades.
2. Aprenderá el manejo de los elementos de la interfaz de Raptor.
3. Aprenderá a integrar instrucciones a programas Raptor.
4. Aprenderá a manejar variables, procesos y estructuras con Raptor.
5. Aprenderá a manejar puntos de interrupción en Raptor.
6. Aprenderá a crear proc edimientos externos en Raptor, y aprenderá cómo consumirlos.
7. Aprenderá a generar gráficos usa ndo las funciones intrínsecas de Raptor.

www.detodoprogramacion.com

Contenido

Raptor como herramienta de desarrollo 210
Capacidades del ambiente de desarrollo 211
Ventana principal (main) 213
Consola maestra (master console) 215
Ejecución de comandos desde master console 216
Instrucciones soportadas 218
Procedimiento de creación de un programa 219
Integrando instrucciones a un programa 222
Declaración de variables y arreglos, y uso de la instrucción assignment 227
Manejo de bucles usando Loop 230
Formas de ejecución de un programa en Raptor y el uso de pruebas de escritorio automáticas 233
Integrando peticiones de datos 235
Integrando condicionales y mostrado de datos 239
Manejo de puntos de interrupción (breakpoints) y comentarios 243
Creación y consumo de subcharts 246
Programación de casos prácticos usando Raptor 251
Caso 1: Mensaje 251
Caso 2: Niveles de servicio 252
Caso 3: Muestra de una tabla de multiplicar 253
Caso 4: Muestra las tablas de multiplicar del 1 al 5 255
Caso 5: Análisis de promedios 257
Desarrollando un programa de regla de tres simple usando Raptor 259
Desarrollando un programa para el cálculo del área de un polígono irregular usando Raptor 259
Desarrollando un programa con bucles, condicionales y arreglos usando Raptor 261
Mapa mental del capítulo 262
Terminología 263
Preguntas 263
Examen rápido 265

i tratas de aprender dos cosas nuevas al mismo tiempo
aprender s una tercera cosa ue no se puede.

Raptor como herramienta de desarrollo

La *cademia de la uerza rea de los Estados nidos* (*nited tates ir orce cademy*) desarrolló el programa *Raptor* (*Rapid lgoritmic Prototyping ool*

for rdered Reasoning), con la finalidad de facilitar la dificultad en el aprendizaje del desarrollo de algoritmos.

Explicar algoritmos utilizando como herramienta de implementación un lenguaje de programación textual, tal como C, Pascal, C , etcétera, ofrece pobres resultados. Por más simple que fuera el lenguaje de programación utilizado, las personas tenían que dividir su atención entre aprender a diseñar algoritmos, y al mismo tiempo tenían que aprender el lenguaje de programación utilizado en la implementación. Esta división de atención, aunque sutil, sabotea todo el proceso de aprendizaje de algoritmos.

En caso de que el algoritmo no funcionara adecuadamente siempre surgía la duda ¿el error está en el algoritmo, o en el programa? Tratar de corregir el problema del algoritmo modificando el programa nunca fue la solución, y tratar se subsanar problemas del programa modificando el algoritmo tampoco. Un proceso de aprendizaje es un proceso de prueba y error; aumentar las posibilidades de error simplemente aumenta la curva de aprendizaje.

El problema de aprender algoritmos utilizando como base lenguajes de programación no es el mayor de los problemas. Lo preocupante es que el no tener la habilidad de corregir un problema rápidamente sólo produce frustración, incertidumbre y caos.

En la actualidad pocas personas son las que encuentran en la frustración continua un motivo para seguir intentando. La mayoría de las personas prefieren mejor admitir que "eso no es lo suyo" y dejan el aprendizaje de los algoritmos por la paz. Más adelante en su vida aprenderán a programar, pero no serán programadores excepcionales hasta que sepan traducir los problemas reales en algoritmos, los algoritmos en soluciones, y las soluciones en programas.

Capacidades del ambiente de desarrollo

Raptor es un lenguaje visual de desarrollo basado en símbolos de diagramas de flujo. En Raptor, cada símbolo del diagrama representa una instrucción específica y las líneas de flujo determinan el orden que debe seguir la ejecución del programa. Al hacer el diagrama de flujo de programa realmente estamos haciendo un programa que podremos ejecutar y comprobar su funcionamiento, lo cual nos evita la necesidad de tener que implementar un diagrama de flujo en un lenguaje de programación.

Al trabajar con Raptor podemos estar en dos tiempos distintos. Estamos en *tiempo de dise o* cuando podemos integrar símbolos a un programa en Raptor; estamos en *tiempo de ejecución* cuando el programa está ejecutándose y produciendo resultados.

Las *caracter sticas de Raptor* son las siguientes.

- *enguaje visual de programación*.

  Raptor es una herramienta de programación visual libre de código, en la cual las instrucciones se proporcionan al integrar en forma de diagrama un conjunto específico de símbolos básicos, de forma ordenada y sintácticamente correcta.

- *erramienta de diagramación*.

  Raptor es una herramienta de diagramación, dado que permite la integración de símbolos interconectados a través de líneas de flujo, que representan un programa. Esta característica es utilizada en tiempo de diseño.

- *onsola de salida*.

  Raptor es también una consola de salida, en la cual se muestran los resultados de los programas que desarrollamos, además de poder ejecutar comandos desde dicha consola sin necesidad de integrar símbolos en el programa. Esta característica es utilizada en tiempo de ejecución.

- *Pruebas de escritorio autom ticas*.

  Raptor posee la capacidad de mostrar el valor que las variables y arreglos contenidos en el programa van tomando en tiempo de ejecución. Esto permite la realización de pruebas de escritorio automáticas. Esta herramienta es utilizada en tiempo de ejecución.

- *eguimiento visual de la ejecución*.

  Raptor permite ver gráficamente la secuencia de ejecución de las instrucciones que componen un programa. En tiempo de ejecución siempre se sabe qué instrucción es la que se está ejecutando, y por qué parte del programa pasa el control del mismo.

- *oporte a funciones*.

  Raptor soporta diversas funciones gráficas, matemáticas y trigonométricas que pueden apoyar el desarrollo de programas complejos.

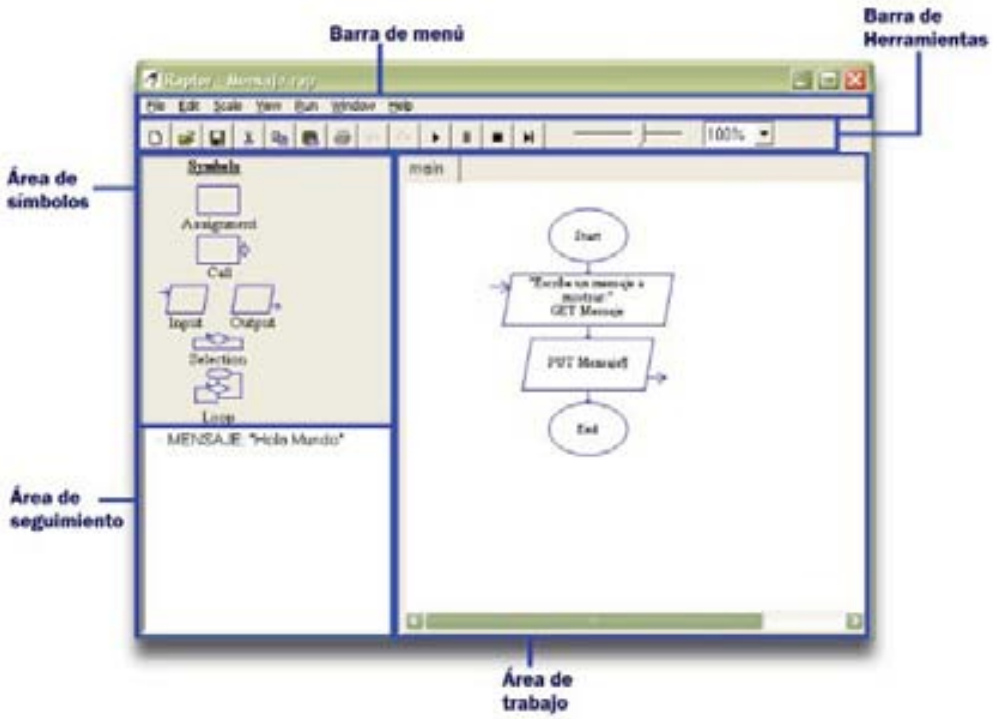
www.detodoprogramacion.com

Ventana principal (main)

La interfaz de Raptor se compone de dos elementos principales: la *ventana principal* (*main*), y la *consola maestra* (*master console*).

Main es la ventana principal de trabajo en tiempo de diseño, y se compone como sigue.

Figura 09.01
Ventana principal de Raptor (main)



En la interfaz de **Main** se tiene los siguientes elementos.

- *arra de men menu bar* .

  Permite la selección de comandos disponibles en **main**. Este elemento es estándar para todos los programas de interfaz indo s.

- *arra de herramientas toolbar* .

  Permite la selección rápida de los comandos más usuales disponibles en Raptor. Este elemento es estándar para todos los programas de interfaz indo s.

Los elementos de toolbar son los siguientes.

Figura 09.02
Barra de herramientas de Raptor (toolbar)

- *rea de s mbolos symbols area .*

  Es el área que posee los 6 símbolos primarios de Raptor para la construcción de programas. Dichos símbolos son los siguientes.

  1. Símbolo de asignación (*assignment symbol*).
  2. Símbolo de llamada (*call symbol*).
  3. Símbolo de entrada (*input symbol*).
  4. Símbolo de salida (*output symbol*).
  . Estructura de selección (*selection structure*).
  6. Estructura de bucle (*loop structure*).

- *rea de seguimiento atch indo .*

  Permite ver los valores contenidos en las variables y arreglos en tiempo de ejecución.

- *rea de trabajo or space .*

  Es el área en donde se pueden integrar de manera visual las instrucciones de Raptor, dando lugar a los programas. En este espacio también se pueden integrar procedimientos externos que complementen a un programa.

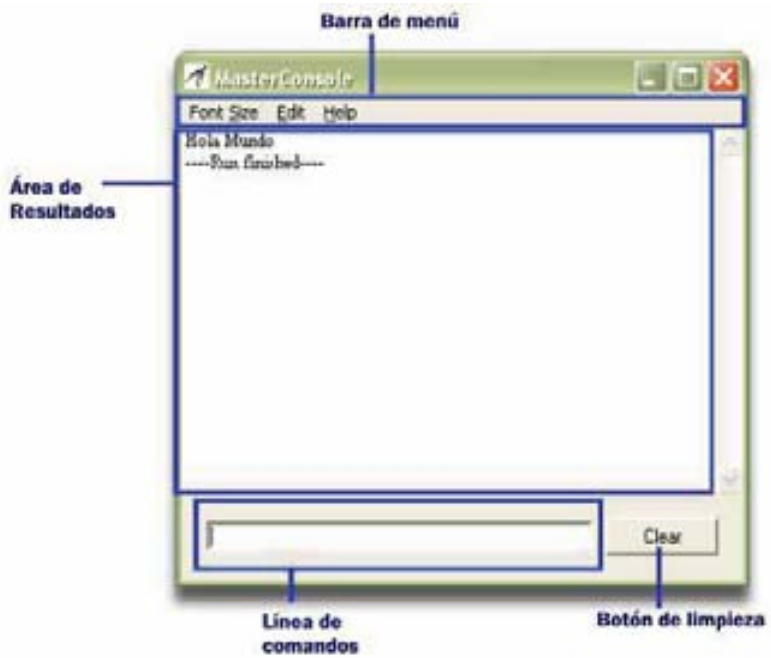
NOTA

Dado que será más fácil encontrar ayuda de Raptor en inglés, nos referiremos a los elementos de Raptor por su término en inglés.

Consola maestra (master console)

Master c onsole es la ventana principal de trabajo en tiempo de ejecución, y luce como sigue.

Figura 09.03
Consola maestra de Raptor (master console)



La interfaz de **master console** se compone de los siguientes elementos.

- *arra de men menu bar .*

  Permite la selección de comandos disponibles en **master con sole**. Este elemento es estándar para todos los programas de interfaz indo s.

- *rea de resultados results .*

  Es el área en la que se mostrarán los resultados de la ejecución del programa.

- *nea de comandos command line .*

  Permite ejecutar comandos directamente desde **master console**.

- *otón de limpiado clear .*

  Limpia el contenido de **results**.

Es importante mencionar que **master console** permanece abierta mientras Raptor esté abierto, y se cierra sólo al cerrar **main**.

NOTA

El instalador de la versión de Raptor utilizada en el presente libro podrá descargarse desde la página Web de Aprenda Practicando, en los recursos del libro. Raptor está escrito en una combinación de los lenguajes Ada, C# y C++, ejecutando bajo .NET Framework. Es necesario que tenga instalado en su equipo el .NET Redistributable Package 1.1 o posterior, para que el programa funcione.

Ejercicio 09.01

Ejecución de comandos desde master console

En este ejercicio se demostrará la forma en que se pueden ejecutar comandos desde *master console*.

1. Abra Raptor, seleccionando *Inicio odos los programas aptor*. Debe ubicar el icono de Raptor en el menú.

2. Se abrirán las dos ventanas principales de Raptor, *main* y *master console*.
3. Seleccione *master console*. Si no puede ubicar la ventana *master console*, presione las teclas Alt ab hasta que ubique la ventana.
4. En la línea de comandos de *master console* escriba los siguientes comandos, y vea lo que sucede. Los comandos se

proporcionan uno a uno, y se dan por ingresados cuando usted presiona ntro. Si el comando está mal escrito o incompleto, Raptor emitirá un pequeño sonido por el altavoz de la computadora, y no hará nada. No se preocupe por las mayúsculas y minúsculas, dado que Raptor no es sensible a ellas (*non case sensitive*).

a) Open_Graph_Window(300,300)

b) Draw_Circle(150,150,100,Red,False)

c) Draw_Circle(150,150,20,Blue,True)

d) Draw_Circle(185,185,10,Blue,True)

e) Draw_Circle(115,185,10,Blue,True)

f) Draw_Arc(100,90,200,140,100,120,200,120,Blue)

g) Close_Graph_Window

Si todo transcurrió correctamente, se habrá dibujado la siguiente figura antes de proporcionar el último comando.

. Cierre el programa Raptor.

FIN DEL EJERCICIO ✱

Instrucciones soportadas

En Raptor las instrucciones son símbolos, que pueden ser integrados a un diagrama de flujo que representa un programa Raptor.

Las instrucciones permitidas por Raptor se encuentran en el área de símbolos que se encuentra en **main**, y son las siguientes.

Símbolo	Instrucción y su aplicación
	Assignment **(*Instrucción de Asignación*)** Permite asignar valores a variables y arreglos. Es el elemento principal para los cambios de estado en un programa.
	Call **(*Instrucción de llamado*)** Permite hacer llamados a funciones de Raptor y a procedimientos externos al programa. En Raptor es posible agregar subrutinas en forma de diagramas, que podrán ser llamadas usando esta instrucción.
	Input **(*Instrucción de entrada*)** Permite realizar peticiones de datos al usuario. En esta instrucción es posible integrar el texto de solicitud (PUT) y el requerimiento del valor solicitado (GET).
	Output **(*Instrucción de salida*)** Permite realizar el mostrado de datos al usuario.
	Selection **(*Instrucción de decisión*)** Permite realizar una estructura de decisión. Dependiendo del valor de una expresión lógica, será posible aplicar instrucciones, tanto si es verdadero (True) como si es falso (False). Esta instrucción permite la implementación de condicionales simples y condicionales complejas.
	Loop **(*Instrucción de bucle*)** Permite realizar una estructura de control. Esta instrucción funciona como bucle de comparación al final. Al cumplirse la condición por verdadero, el ciclo termina; como en todos los bucles de comparación al final, se ejecuta al menos una vez.

Es fundamental que entienda que Raptor es un lenguaje visual. Lo que usted genera parece un diagrama, pero en realidad es una secuencia de instrucciones que tienen sintaxis y reglas sintácticas. Gracias al enfoque visual de Raptor no es posible cometer errores de sintaxis, lo que aporta mucha certidumbre al desarrollador.

Procedimiento de creación de un programa

El procedimiento para la creación de programas en Raptor es muy simple. Basta con desarrollar el diagrama de flujo de forma correcta, y el programa ya estará realizado.

La forma más práctica de aprender a programar es programando, por lo que trataremos el procedimiento en forma de ejercicios. Los puntos a tratar en el procedimiento son los siguientes.

1. Integración de instrucciones.
2. Declaración de variables y arreglos.
3. Petición y entrada de datos.
4. Manejo de condicionales.
. Manejo de bucles.
6. Llamado a funciones.
. Diseño y consumo de subrutinas.

El programa que utilizaremos para demostrar el desarrollo en Raptor es el siguiente.

Adivina

Se pide elaborar un juego de adivinanzas.

El juego consiste en tratar de adivinar números que la computadora ha generado de forma aleatoria.

El programa generará 5 números aleatorios, cada uno entre el 1 y el 3, mismos que no revelará. Posteriormente, le preguntará al usuario 5 números, cada uno entre el 1 y el 3. Los aciertos se cuentan considerando las coincidencias de números, por ejemplo, si el primer número generado por la computadora es igual al primer número proporcionado por el usuario, entonces se ha obtenido un acierto, de lo contrario, se tiene un fallo. Cada vez que el número generado por la computadora coincida con el proporcionado por el usuario se tendrán aciertos.

En caso de que el usuario haya acertado al menos a dos de los números generados por la computadora, habrá ganado el juego. Como premio, aparecerá una carita feliz.

Como medida de transparencia en el juego, el programa deberá mostrar, número por número, si se tuvo acierto o fallo.

www.detodoprogramacion.com

A continuación mostramos un juego ganador.

Computadora	Jugador
2	1
3	**3**
1	3
1	**1**
2	1

Como se acertaron 2 números (posiciones 2 y 4), el jugador gana.

A continuación mostramos un juego perdedor.

Computadora	Jugador
2	1
3	1
1	3
1	**1**
2	1

Como se acertó 1 número (posición 4), el jugador pierde.

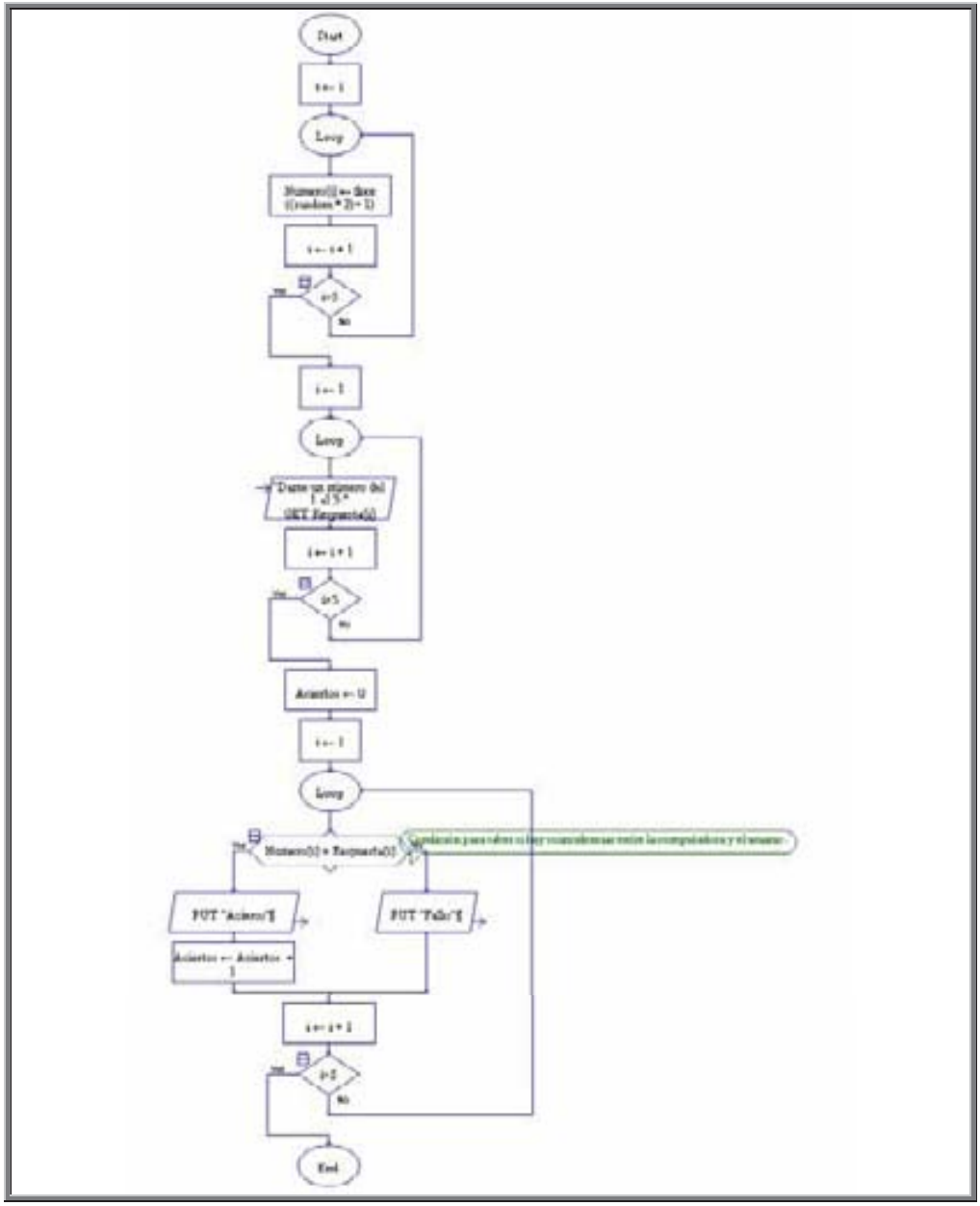
El algoritmo del programa es el siguiente.

Algoritmo orientado a datos

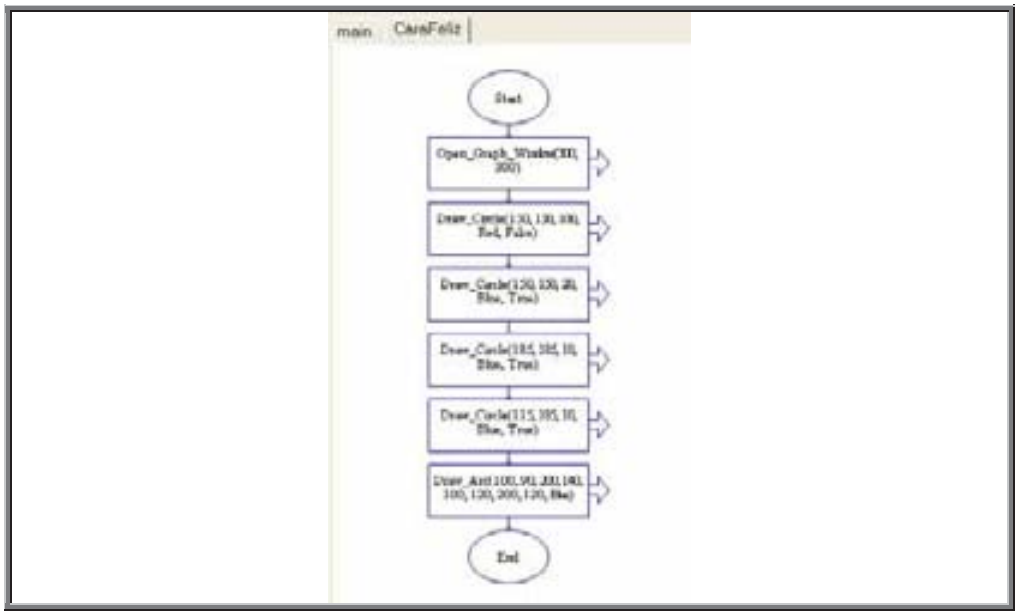
1. Inicio.
2. Se generan 5 números aleatorios del 1 al 3 (Numero(5),b[1-3]).
3. Se preguntan al usuario 5 números del 1 al 3 (Respuesta(5),b[1-3]).
4. Se comparan los números de la computadora (Numero) y los del usuario (Respuesta).
5. Se informa si el número generado por la computadora es igual al proporcionado por el usuario (acierto) o no (fallo).
6. Se cuentan los aciertos (Aciertos,b[0-5]).
7. Si se tienen dos o más aciertos, mostrar una carita feliz.
8. Fin.

Al haber finalizado todos los ejercicios, se habrán creado los siguientes programas en Raptor.

Adivinanza.rap



Adivinanza.rap - CaraFeliz



Ejercicio 09.02

Integrando instrucciones a un programa

En el presente ejercicio veremos cómo se integran instrucciones a un programa en Raptor. Se agregarán las instrucciones para cubrir el primer paso del algoritmo, que consiste en generar 5 números aleatorios y almacenarlos en variables de memoria.

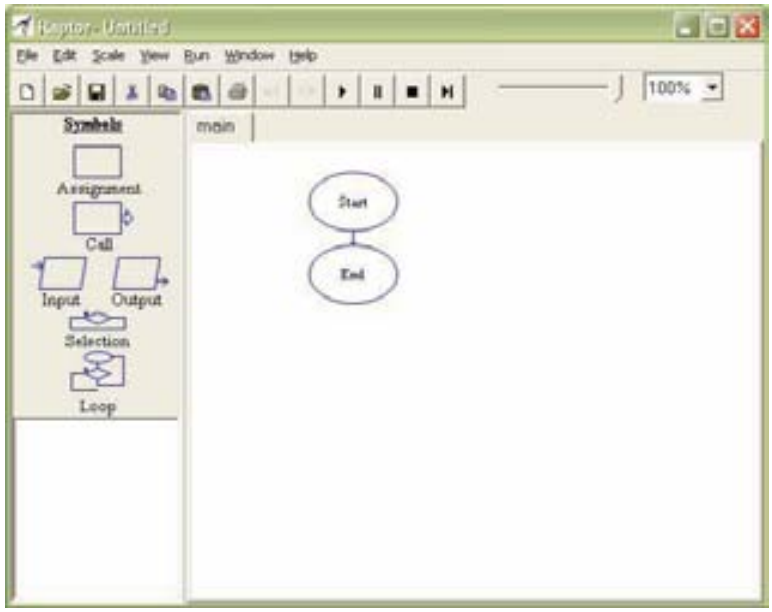
Miniespecificación de esta parte del programa

```
1   INICIO:
2           (i,i[0-5])
3           (Numero(5),i[1-3])
4           i ← 1
5           Inicio
6                   Numero(i) ← (i,[1-3] Random)
7                   i ← i + 1
8           Hasta que i > 5
9   ...
```

1. En caso de que no tenga abierto Raptor, seleccionando *Inicio odos los programas aptor*. Debe ubicar el icono de Raptor en el menú.

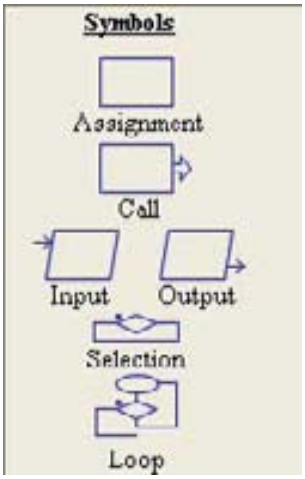
2. Es probable que en cualquier momento Raptor le solicite que guarde su programa por primera vez. Cuando eso suceda, asigne como nombre de programa Adivinanza.rap. Como debe suponer los programas en Raptor tienen la extensión RAP.

3. Aparecerá la interfaz de Raptor, y en *or space* aparecerá un programa nuevo, con las instrucciones de inicio y final. En Raptor es imposible cometer un error por omisión de inicio y fin, ya que estos símbolos siempre estarán presentes.



4. Vea cómo entre la instrucción de inicio (**Start**) y la instrucción de fin (**End**), hay una **línea de flujo**, que posee cabeza de flecha. La línea indica que las instrucciones están interconectadas, pero además indica que el sentido del programa es Top Do n.

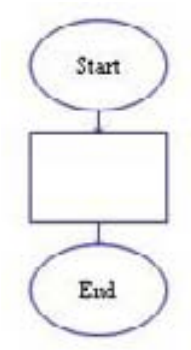
. **Agregado de instrucciones usando symbols area.** Hay dos formas de agregar instrucciones a un programa Raptor. La primera consiste en hacer clic sobre alguna de las 6 instrucciones que aparecen en *s m ols area*.



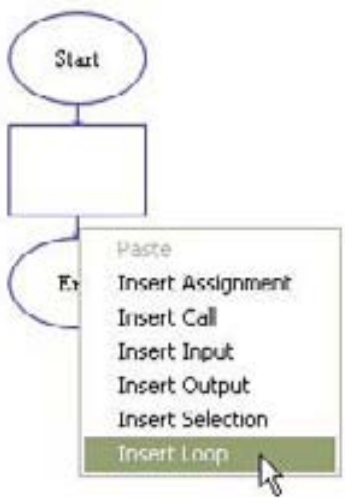
6. Al hacer clic en una de las instrucciones, esta se pondrá en color rojo, indicando que es el tipo de instrucción que se agregará. Seleccione la instrucción **assignment** haciendo un solo clic sobre el símbolo correspondiente que se encuentra en *s m ols area*.

. Ahora coloque el puntero del ratón sobre la línea de flujo que se encuentra entre **Start** y **End**. Observe cómo al colocar el puntero del ratón sobre la línea, éste cambia de aspecto, luciendo como una pequeña manita.

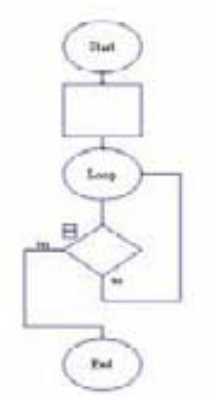
. Cuando el puntero tiene ese aspecto y se hace clic sobre la línea, la instrucción se agregará al diagrama. Haga clic sobre la línea de flujo entre **Start** y **End**, cuando el puntero del ratón luzca como una manita, y la instrucción se agregará.



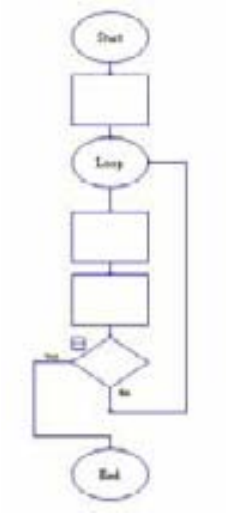
. **Agregado de i nstrucciones usa ndo menú contextual.** Otra forma de agregar instrucciones es utilizando el menú contextual. Coloque el puntero del ratón en la línea de flujo entre **assignment** y **End**. Aparecerá el puntero en forma de manita. Presione el botón derecho del ratón, y aparecerá el menú contextual con todas las opciones de agregado de instrucciones. Seleccione la opción de menú contextual *Insert oop*.



10. Se debió haber agregado una instrucción Loop al diagrama.



11. Agregue dentro de la estructura **Loop** dos **assignment** más, de tal forma que el diagrama quede como sigue. Utilice el método que mejor le parezca para la integración de las dos instrucciones.



12. Grabe su programa, utilizando cualquiera de las formas de guardado: menú *ile ave*, trl , o haciendo clic en *uardar diagrama* de *ool ar*.

FIN DEL EJERCICIO ✱

Ejercicio 09.03

Declaración de variables y arreglos, y uso de la instrucción assignment

En el presente ejercicio veremos la forma en que se declaran variables y arreglos en Raptor, y la forma en que se realizan cambios de estado. Seguiremos trabajando con Adivinanza.rap.

1. En Raptor se tiene la posibilidad de manejar variables y arreglos. La forma de declararlos es mediante un **assignment**, a través del cual se les otorga un valor inicial. Ninguna variable o arreglo puede ser utilizado como expresión si antes no se le ha asignado un valor inicial, ya sea mediante **assignment** o mediante una petición de datos realizada a través de **input**.

2. Para declarar una variable se utiliza la instrucción **assignment**. Haga doble clic sobre el **assignment** que se encuentra entre **Start** y **Loop**. Aparecerá la ventana *Enter tatement*. En *et* debe colocarse el nombre de la variable a declarar, y en *to*, el valor a asignar. Si la variable ya fue creada con anterioridad, esta ventana servirá para hacer cambios de estado.

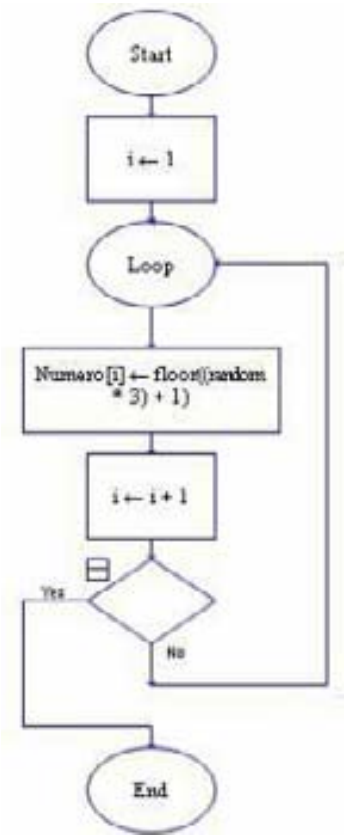
3. Como puede ver, en la *Enter tatement* hay mucho espacio que parece inutilizado. Ese espacio es utilizado por Raptor para proponer funciones predefinidas en caso de que detecte que podemos estar interesados en alguna de ellas. Escriba en *to* el valor Is_ , y vea lo que sugiere Raptor. Esta facilidad está disponible en todas las ventanas en donde se involucren valores y vea espacios que en apariencia no se aprovechan.

4. En *Enter tatement* especifique en *et* el nombre de variable i, y en *to* especifique el valor de 1. Haga clic en *one* para dar por aceptado el cambio de estado.

. En Raptor se pueden definir arreglos de 1 y 2 dimensiones. El número de elementos es dinámico, es decir, no se especifica. La definición de arreglos se obtiene al agregar a un nombre de variable un subíndice, encerrado entre corchetes "[ ]". Por ejemplo, para tener acceso al elemento 4 de un arreglo llamado Numero, tendríamos que especificar Numero[4]. Si el arreglo fuera de dos dimensiones y si queremos el dato de la fila 4, columna 5, tendríamos que especificar Numero[4,5].

6. Haga clic sobre el segundo **assignment** del programa. En *Enter tatement* especifique en *et* el nombre de variable Numero[i], y en *to* especifique el valor de floor((random*3)+1). El valor que se está asignando utiliza las funciones floor y random de Raptor para construir un número aleatorio del 1 al 3. Haga clic en *one* para dar por aceptado el cambio de estado.

. Haga clic sobre el tercer **assignment** del programa. En *Enter tatement* especifique en *et* el nombre de variable i, y en *to* especifique el valor de i + 1. Se está definiendo un contador. En caso de no haber declarado la variable i en el primer **assignment**, el programa generaría error, dado que estaríamos pretendiendo utilizar una variable que no ha adquirido valor aún. Haga clic en *one* para dar por aceptado el cambio de estado. Nuestro programa luciría como sigue.



. Grabe su programa, utilizando cualquiera de las formas de guardado: menú *ile ave*, trl , o haciendo clic en *uardar diagrama* de *ool ar*.

FIN DEL EJERCICIO ✱

Ejercicio 09.04

Manejo de bucles usando Loop

En el presente ejercicio veremos la forma en que implementa un bucle, de tipo bucle de comparación al final. Trabajaremos con Adivinanza.rap.

1. En Raptor se pueden habilitar bucles de comparación al final utilizando **Loop**. La forma de implementarlo es la siguiente.

    a) Se requiere una instrucción **assignment**, una instrucción **Loop**, y un **assignment** dentro de **Loop**.

    b) Utilizando **assignment** se define inicialmente una variable de control, generalmente un contador.

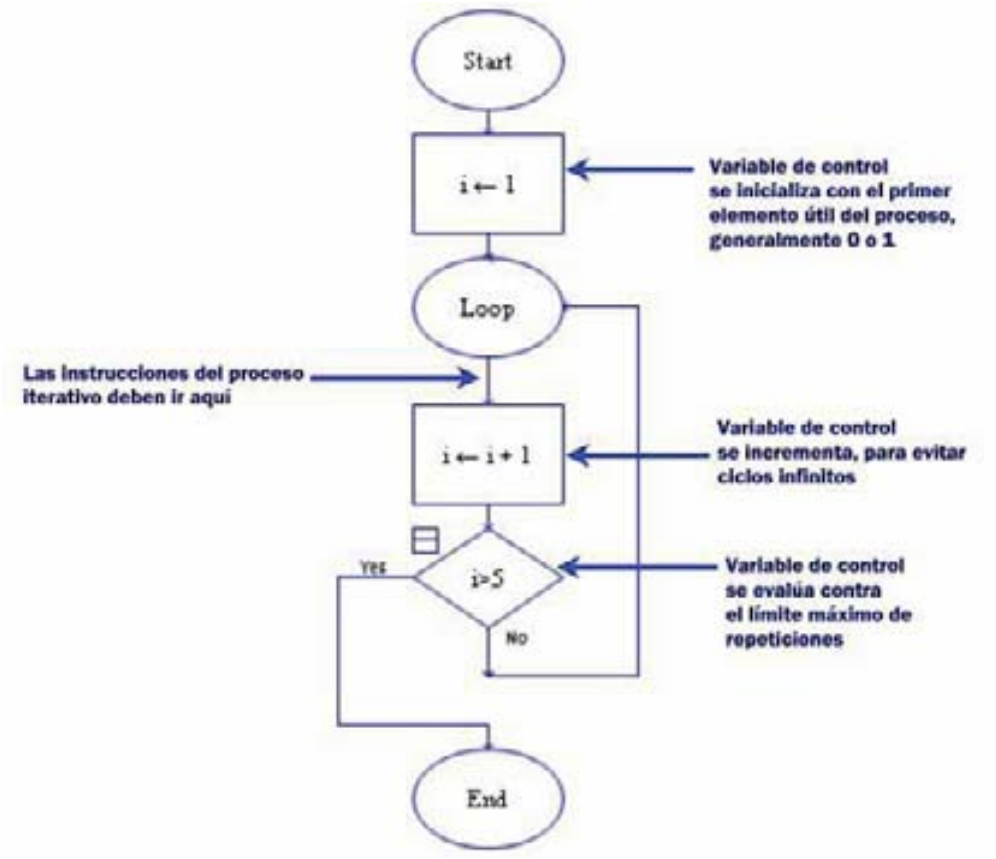
    c) Se inicia el bucle usando **Loop**, que culmina con una condicional, llamada *loop condition*, que deberá evaluar la variable de control.

    d) Para evitar la creación de un ciclo infinito, la variable de control debe cambiar su estado cada vez que el ciclo se repite. Generalmente esto equivale a incrementar un contador.

    e) Se sugiere que el cambio de estado se realice de forma adyacente a **loop condition** ; obviamente, dentro del ciclo. Si la condición se resuelve por verdadero, el proceso se termina.

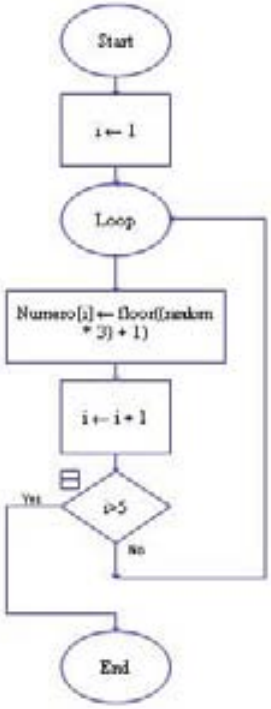
    f) Las instrucciones que se han de repetir en el ciclo deben colocarse bajo el símbolo de inicio del **Loop**.

    g) A continuación se ilustra un bucle que se repite 5 veces.



2. Para implementar nuestro bucle en Raptor sólo nos hace falta agregar la condición de evaluación del bucle. Haga doble clic en la condicional que aparece al final del ciclo. Con ello aparecerá la ventana *Enter oop ondition*. En ella podrá especificar la condición de evaluación.

3. Ingrese como condición i>5, dado que queremos que el ciclo se repita del número en que se inició el contador (1) hasta 5. Haga clic en *one* para aceptar los cambios.
4. Nuestro programa debe lucir como sigue.



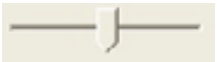
. Grabe su programa, utilizando cualquiera de las formas de guardado: menú *ile ave*, trl , o haciendo clic en *uardar diagrama* de *ool ar*.

FIN DEL EJERCICIO ✱

Ejercicio 09.05

Formas de ejecución de un programa en Raptor y el uso de pruebas de escritorio automáticas

En el presente ejercicio veremos cómo se ejecuta un programa Raptor. Veremos también cómo podemos aprovechar *atc indo* para las pruebas de escritorio automáticas. Comprobaremos también la capacidad de Raptor de mostrar visualmente la ejecución del programa. Seguiremos trabajando con Adivinanza.rap.

1.  **Control de velocidad de ejecución.** El *control de velocidad de e ecuci n* que aparece en *ool ar* permite especificar la velocidad en la que desea que el programa se ejecute. Si arrastra *el indicador de velocidad* hasta el extremo derecho de la línea, el programa ejecutará sin que usted vea la secuencia visual de ejecución que integra Raptor.

2. Arrastre el indicador de velocidad del *control de velocidad de e ecuci n* hasta la extrema derecha, a fin de que la ejecución sea lo más rápida posible.

3. **Tipos de e jecución.** Raptor tiene tres formas de ejecución: *ejecución del programa completo*, *ejecución instrucción por instrucción* y *ejecución compilada*.

4. La ejecución del programa completo se logra al presionar **5** o seleccionar *un eset E ecute*.

. La ejecución instrucción por instrucción se logra al presionar **1**0 o seleccionar *un tep*.

6. En cualquier momento se puede detener la ejecución de un programa al seleccionar *un ause*.

. En cualquier momento puede darse también por terminada la ejecución del programa desde **Toolbar**.

. La ejecución compilada se utiliza al seleccionar *un un compiled*. En esta modalidad no se interactúa con *main* ni se obtienen las ventajas de seguimiento. Sólo se obtiene la salida en *master console*.

. **Ejecución de programa completo.** Presione **5** y vea lo que sucede.

10. Inmediatamente debió aparecer la ventana que indica la conclusión de la ejecución.

11. **Seguimiento a las variables usando watch window.** La herramienta *atc indo* que se encuentra en *main* permite visualizar el estado de las variables en todo momento.

12. Para que el contenido del arreglo se muestre, debe hacer clic en el símbolo "+" que aparece al lado del nombre del arreglo. Vea los valores finales que asumieron las variables y arreglos del programa. La variable i, desde luego, tiene un valor de 6, es decir > 5, que fue lo que provocó que el programa se detuviera. Se muestra también el arreglo Numero. *atc indo* nos dice que el tamaño del arreglo es de 5 elementos, muestra el subíndice de cada uno de los elementos y los valores que de forma aleatoria la computadora asignó a cada elemento. Como puede ver, los valores están en un rango del 1 al 3.

13. **Muestra visual del flujo de ejecución.** Ahora arrastre el *indicador de velocidad* del *control de velocidad de ejecución* hasta el extremo izquierdo de la línea, a fin de que la ejecución sea lo más lenta posible.

14. Presione **5** para que vea cómo Raptor ilustra el flujo de ejecución del programa en forma visual. Esta utilidad es muy valiosa para ver de qué forma se comporta nuestro programa, por lo que es muy fácil detectar cuando el programa toma un flujo que no esperamos.

15. **Ejecución instrucción por instrucción.** Seleccione en el diagrama la instrucción **Start**. Deberá cambiar su color a rojo.

16. Presione consecutivamente **10**, para ejecutar el programa instrucción por instrucción. Vea el comportamiento *de atcindo*.

17. Cuando aparezca NUMERO en *atcindo*, haga clic en el símbolo "+ " para ver el detalle de lo que pasa con el arreglo.

18. No deje de presionar **10** hasta que el elemento 3 del arreglo haya tomado valores.

19. Como puede ver, Raptor ofrece mucha funcionalidad, tanto para dar seguimiento a la ejecución del programa, como a los valores que van tomando las variables.

20. Grabe su programa, utilizando cualquiera de las formas de guardado: menú *ile ave*, trl , o haciendo clic en *uardar diagrama* de *ool ar*.

FIN DEL EJERCICIO ✱

Ejercicio 09.06

Integrando peticiones de datos

En el presente ejercicio veremos cómo se integran peticiones de datos en Raptor. Como guía utilizaremos la segunda parte de la miniespecificación de nuestro ejemplo. a la computadora sabe sus números y ahora el usuario deberá proporcionar los suyos. Seguiremos trabajaremos con Adivinanza.rap.

Miniespecificación de esta parte del programa

```
9          i ← 1
10         Inicio
11                 « "Dame un número del 1 al 3: "
12                 » Respuesta(i)
13                 i ← i + 1
14         Hasta que i > 5
15   ...
```

1. **Peticiones de datos.** La petición de datos en Raptor se logra utilizando la instrucción **input**. Al agregar la instrucción y hacer doble clic en símbolo aparece la ventana *Enter Input*.

2. Raptor proporciona la facilidad de integrar en una sola instrucción el mensaje de solicitud (PUT) y la obtención del dato por parte del usuario (GET), por lo que las líneas 11 y 12 de la miniespecificación se cubren con una sola instrucción de Raptor.

3. En *Enter Input* se especifican dos cosas: el mensaje de solicitud del dato y la variable en la que el dato se almacenará.

4. En la parte superior de la ventana aparecen unos botones de radio que solicitan especificar si el mensaje de solicitud es una expresión (*E pression*) o texto (*e t*); el mensaje de solicitud de dato se deberá especificar en el primer cuadro de texto.

. Si se trata de una expresión todas las cadenas deben ir entre comillas y la expresión debe ser válida; si se trata de texto, todo

será considerado texto y no se requieren comillas. Tenga cuidado: si coloca un texto pero selecciona *E pression*, Raptor tratará al texto como una variable y generará error. Utilice *E pression* cuando el texto de solicitud del dato requiere construirse de varias expresiones (expresión compuesta), y utilice *e t* cuando el texto de solicitud del dato es simple.

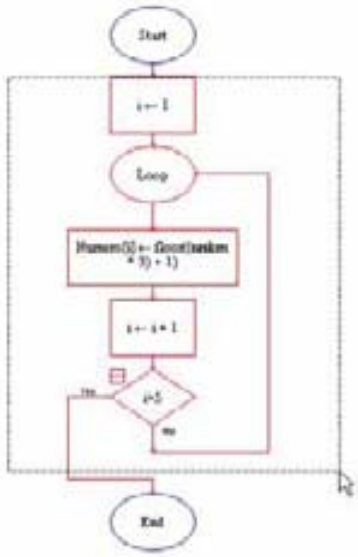
6. En el segundo cuadro de texto deberá colocarse el nombre de la variable o elemento de arreglo que ha de recibir el valor. Si la variable no está declarada con anterioridad, Raptor la declarará al momento en que se le proporcione información por primera vez.

. Continúe donde nos quedamos en el programa. Las instrucciones se agregarán justo antes de la instrucción **End**.

. **Reutilización de instrucciones.** Agregue al final del programa un nuevo bucle que repita una operación 5 veces.

. Recuerde que los diagramas son programas, y como tales aceptan copiar y pegar instrucciones. En nuestro programa ya tenemos un bucle del 1 al 5, mismo que podemos reutilizar.

10. En *or space* trace un cuadro imaginario que alcance todo el bucle definido anteriormente. Todas las instrucciones entre **Start** y **End** estarán en rojo, lo que indica que están seleccionadas.



11. Ahora copie las instrucciones seleccionadas, presionando trl o seleccionando *Edit op* .

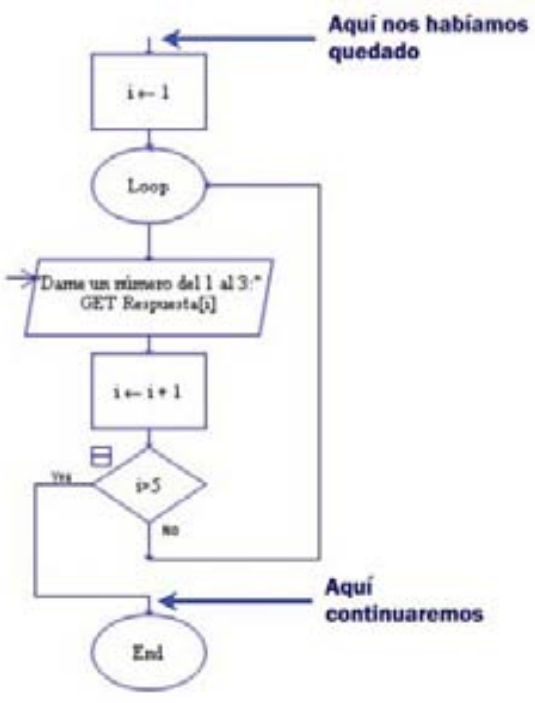
12. Ahora coloque el puntero del ratón justo sobre la línea de flujo que aparece sobre la instrucción **End**. El puntero debe tener la forma de una manita. Utilice el menú de contexto y seleccione *aste*.

13. **Eliminando i nstrucciones.** Al hacer esto se pegarán las instrucciones que forman el bucle. No todo lo que copiamos nos sirve. La instrucción **assignment** que coloca números aleatorios en el arreglo no tiene razón en el segundo bucle. Haga clic en el **assignment** Numero[i] ← floor((random*3)+1), y presione la tecla s ri ir, para eliminar la instrucción.

14. Es posible seleccionar más de una instrucción y eliminarla.

1 . Dentro del segundo bucle agregue una instrucción **input**.

16. Haga doble clic sobre el **input** y en *Enter Input* seleccione *e t*; escriba en el primer cuadro de texto Dame un número del 1 al 3, y en el segundo cuadro de texto escriba el nombre de la variable Respuesta[i]. La parte final del programa debe quedar como sigue.



1 . Ejecute su programa presionando **5**. Ahora el programa deberá pedirle 5 veces que le proporcione un número del 1 al 3.

1 . Al finalizar el programa, utilice *atc indo* para ver en cuántas ocasiones los números que la computadora tiene (NUMERO) corresponden a los números que usted proporcionó (RESPUESTA). Para que un número cuente como acierto, debe tener el mismo número del 1 al 3 en el mismo subíndice de NUMERO y RESPUESTA.

1 . Grabe su programa, utilizando cualquiera de las formas de guardado: menú *ile ave*, trl , o haciendo clic en *uardar diagrama* de *ool ar*.

FIN DEL EJERCICIO ✱

Ejercicio 09.07

Integrando condicionales y mostrado de datos

En el presente ejercicio veremos cómo se integran condicionales compuestos y cómo se hace mostrado de datos. Como guía utilizaremos la tercera parte de la miniespecificación de nuestro ejemplo. a el programa sabe los números de la computadora y los del usuario, por lo que ahora sólo resta comparar. Seguiremos trabajando con Adivinanza.rap.

Miniespecificación de esta parte del programa

```
15          Aciertos ← 0
16          i ← 1
17          Inicio
18                  Si Numero(i) = Respuesta(i) Entonces
19                          « "Acierto"
20                          Aciertos ← Aciertos + 1
21                  De lo contrario
22                          « "Fallo"
23                  Fin Si
24                  i ← i + 1
25          Hasta que i > 5
26  ...
```

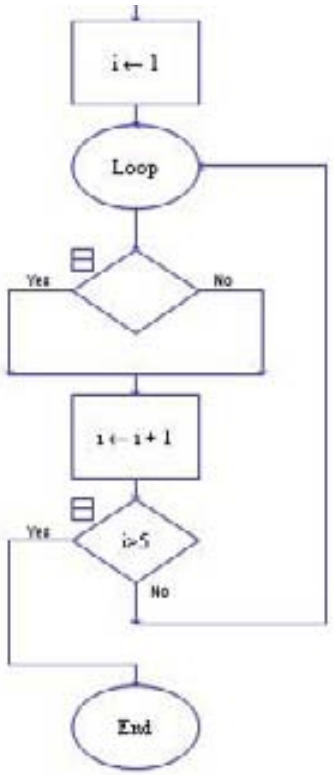
1. Necesitamos otro bucle de 5 repeticiones. Repita la operación de copiado y pegado que realizó en el ejercicio anterior. De igual forma, elimine la instrucción **assignment** Numero[i] ← floor((random*3)+1), porque no la necesitamos.

2. **Agregando una condi cional.** Para agregar condicionales sólo hace falta integrar al programa una instrucción **selection**,

especificar la condición de evaluación, y agregar las instrucciones que queremos que se ejecuten, tanto si la condición resuelve por verdadero (True Yes) o por falso (False No).

3. Agregue antes del bucle una instrucción **assignment** que declare la variable que ha de servirnos para contar el número de aciertos que tenemos. En **assignment** especifique en *et* la variable Aciertos, y en *to* proporcione cer0.

4. Agregue dentro del último bucle una instrucción **selection**. Al hacerlo, la parte final de nuestro diagrama debe quedar como sigue.



. El símbolo está representado por un rombo y dos salidas. En el rombo se deberá colocar la condición a ser evaluada, a lo que se conoce como *condición de selección* (*selection condition*). En la salida que indica Yes (*salida por verdadero*) deberán colocarse las instrucciones que aplican en caso de que la condición resuelva por verdadero, mientras que en la salida que indica No (*salida por falso*) deberán colocarse las instrucciones que aplican en caso de que la condición resuelva por falso.

6. Haga doble clic sobre el rombo de **selection**. Aparecerá la ventana *Enter election ondition*, en la cual se puede especificar una expresión lógica o condición. Esta condición es la que determina el flujo que seguirá el programa. Coloque en el cuadro de texto la condición Numero[i] = Respuesta[i]. Haga clic en *one* para aceptar los cambios.

. **Mostrado de datos.** El mostrado de datos en Raptor se realiza utilizando la instrucción **output**. Agregue una instrucción **output** tanto en la salida por verdadero como en la salida por falso.

. Haga doble clic sobre la instrucción output que colocó en la salida por verdadero. Aparecerá la ventana *Enter utput*, que es donde se especificará el tipo de salida que se desea.

. En *Enter utput* se puede especificar el tipo de salida que se desea. Si la salida es una expresión compuesta deberá seleccionarse *utput E pression*. Si la salida es una expresión simple, seleccione *utput e t*. En el cuadro de texto deberá escribirse la expresión o texto que desea mostrarse.

10. Lo que se establezca como **output** aparecerá en *master console* como salida. En caso de ejecutar la versión compilada del programa, solamente podremos ver aquello que se haya especificado con instrucciones **output**.

11. La casilla de verificación *End current line* permite especificar que la salida debe incluir un salto de línea al final. Si no está marcada la casilla, la salida saldrá en una sola línea.

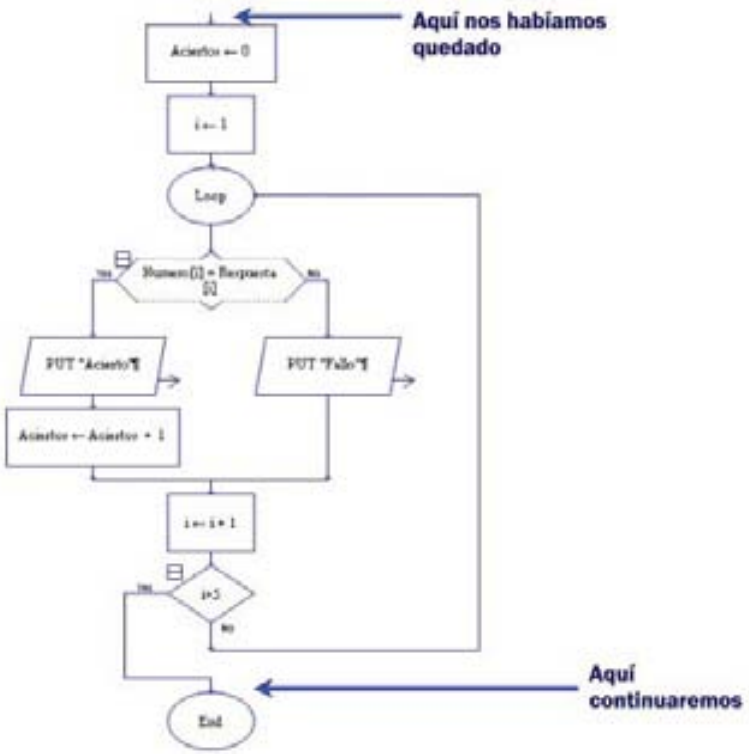
12. Seleccione *utput e t*. En el cuadro de texto escriba Acierto. Aunque se trata de un texto no es necesario encerrarlo entre comillas dobles, dado que especificamos *utput e t*. Asegúrese que la casilla de verificación *End current line* esté activada, y presione *one* para aceptar los cambios.

13. Haga doble clic sobre la instrucción output que agregó en la salida por falso. Aparecerá *Enter utput*.

14. Seleccione *utput e t*. En el cuadro de texto escriba Fallo. Asegúrese que la casilla de verificación *End current line* esté activada, y presione *one* para aceptar los cambios.

1 . Después de la instrucción **output** de la salida por falso, todavía dentro de la estructura de decisión, agregue una instrucción **assignment**, en donde incremente el valor de la variable Aciertos en 1. Para ello haga doble clic en **assignment**. En *et* coloque el nombre de la variable Aciertos, y en *to*, coloque Aciertos + 1. Presione *one* para aceptar.

La parte final de nuestro programa habrá quedado como sigue.



16. Ejecute el programa presionando **5**. Proporcione 5 veces el número 1, y vea si gana o no. Recuerde que para ganar debe tener al menos dos aciertos. *aster console* emitirá la salida del proceso en una forma parecida a la siguiente.

1 . En nuestra prueba, el primero y el último número coincidieron, por lo que se gana el juego.

1 . Grabe su programa, utilizando cualquiera de las formas de guardado: menú *ile ave*, trl , o haciendo clic en *uardar diagrama* de *ool ar*.

FIN DEL EJERCICIO ✱

Ejercicio 09.08

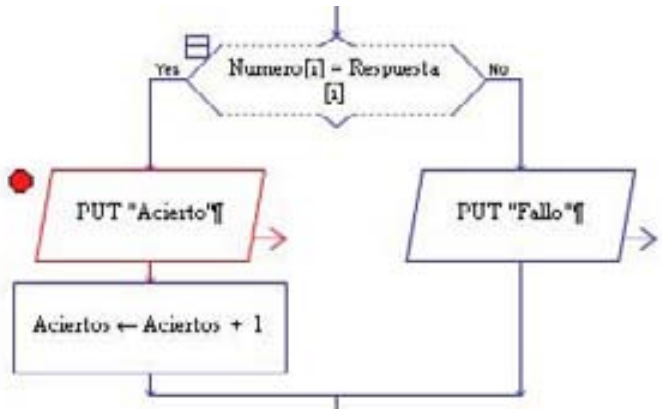
Manejo de puntos de interrupción (breakpoints) y comentarios

En el presente ejercicio veremos el uso de puntos de interrupción (brea points) y comentarios en Raptor. Seguiremos trabajando con Adivinanza.rap.

1. **Agregando puntos de interrupción.** En Raptor es posible agregar *puntos de interrupción*, llamados también *brea points*. Un punto de interrupción es una marca que se coloca al programa para que se detenga al llegar a una determinada instrucción.

2. En Raptor la forma de agregar un punto de interrupción es colocándonos sobre una instrucción e invocando el menú contextual, donde seleccionaremos la opción *oggle rea point*.

3. Seleccione la instrucción output que agregó en la salida por verdadero de la instrucción **selection**, en el ejercicio anterior. Invoque el menú contextual y seleccione la opción *oggle rea point*. Deberá agregarse un punto rojo, luciendo como sigue.



4. ¿En qué elemento tuvo su primer acierto? La respuesta será muy sencilla. Si corremos el programa éste se detendrá en la instrucción que tiene el punto de interrupción. Como dicha instrucción se encuentra dentro de la salida por verdadero de la selección quiere decir que se tuvo un acierto. Bastará con ver qué valor tiene la variable i en *atc indo* . Restándole 1 a i tendremos la respuesta a nuestra pregunta.

. Dado el caso, el programa se detendrá, y podrá continuar con él presionando **5** o **10**, dependiendo la forma en la que desee continuar.

6. Ejecute su programa presionando **5**, proporcione 5 números 1 y determine en qué número obtuvo su primer acierto.

. **Eliminando puntos de interrupción.** Para eliminar los puntos de interrupción basta con seleccionar la opción de menú *un lear all rea points*.

. Elimine los puntos de interrupción del diagrama.

. **Agregando coment arios.** En ocasiones se puede desear agregar comentarios al diagrama. En Raptor se pueden agregar comentarios a las instrucciones. Para ello se debe seleccionar la instrucción a comentar, invocar el menú contextual y seleccionar la opción *omment*. Con ello se mostrará la ventana *Enter omment*.

10. El comentario que se anote en esta ventana estará asociado a la instrucción sobre la cual se invocó el menú contextual.
11. Seleccione la instrucción **selection** del programa de ejemplo. Invoque el menú contextual y seleccione *omment*. En la ventana *Enter omment* escriba Condición para saber si hay coincidencias entre la computadora y el usuario.
12. Para eliminar comentarios basta con seleccionarlos y presionar la tecla *uprimir*.
13. Los comentarios lucen como sigue.

14. Grabe su programa, utilizando cualquiera de las formas de guardado: menú *ile ave*, trl , o haciendo clic en *uardar diagrama* de *ool ar*.

FIN DEL EJERCICIO ✱

Ejercicio 09.09

Creación y consumo de subcharts

En el presente ejercicio veremos cómo se genera un procedimiento externo y cómo se puede llamar a ejecución. Seguiremos trabajando con Adivinanza.rap.

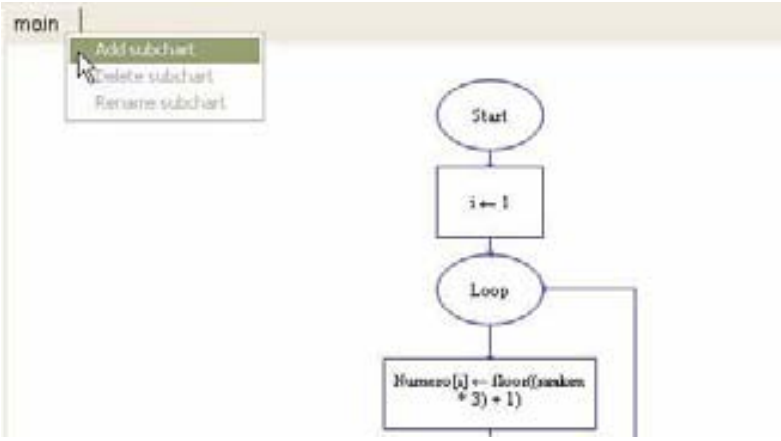
Miniespecificación de esta parte del programa

```
26          Si Aciertos >= 2 Entonces
27                      Ejecutar CaraFeliz
28          Fin Si
29  :FIN
```

1. **Llamadas.** Al iniciar un programa en Raptor, automáticamente en *or space* aparece el *procedimiento principal del programa*, a lo que conocemos como *main*. Todos los procedimientos que no forman parte de *main* pero que pueden ser invocados para aportar su funcionalidad se llaman *procedimientos e ternos*.
2. En Raptor se tienen dos tipos diferentes de procedimientos externos, las *funciones intr nsecas*, como por ejemplo (Open_Graph_Window o ACos), y los *subdiagramas*, conocidos también como *subcharts*.
3. Los *su c arts* son procedimientos desarrollados en Raptor que están asociados a un procedimiento principal que los puede llamar.
4. En Raptor las llamadas a procedimientos externos se hacen utilizando la instrucción **Call**.

. Mediante **Call** se pueden llamar a ejecución funciones intrínsecas y los *su c arts*, sólo basta hacer la referencia a ellos, proporcionando los valores de entrada que requieran, en caso de que aplique.

6. **Creando un subchart.** La forma de crear un *su c art* es haciendo clic en la pestaña *main*, invocar el menú de contexto y seleccionar la opción *dd su c art*.



. Al seleccionar dicha opción aparecerá la ventana *ame u c art*, que permite darle nombre al *su c art* creado, mismo que se dará por aceptado al hacer clic en .

. Para nombrar un *su c art* siga las mismas reglas de nombrado para las variables.

. Agregue en el programa Adivinanza.rap un *su c art* de nombre CaraFeliz. Con ello aparecerá una nueva pestaña en *or space*. Usted podrá estar alternando entre el procedimiento principal (*main*) y el *su c art* con sólo hacer clic en la pestaña de cada uno.

10. Agregue al *su c art* CaraFeliz 6 instrucciones **Call**. Reproduciremos las mismas instrucciones que colocamos en el ejercicio 09.01. Para especificar la función o *su c art* a llamar desde **call** es necesario hacer doble clic sobre la instrucción **call**, con lo que aparecerá la ventana *Enter all*. En el cuadro de texto de la ventana se deberá especificar la función intrínseca o *su c art* a ejecutar.

En el primer **call** colocaremos la siguiente instrucción.

```
Open_Graph_Window(300,300)
```

En el segundo **call** colocaremos la siguiente instrucción.

```
Draw_Circle(150,150,100,Red,False)
```

En el tercer **call** colocaremos la siguiente instrucción.

```
Draw_Circle(150,150,20,Blue,True)
```

En el cuarto **call** colocaremos la siguiente instrucción.

Draw_Circle(185,185,10,Blue,True)

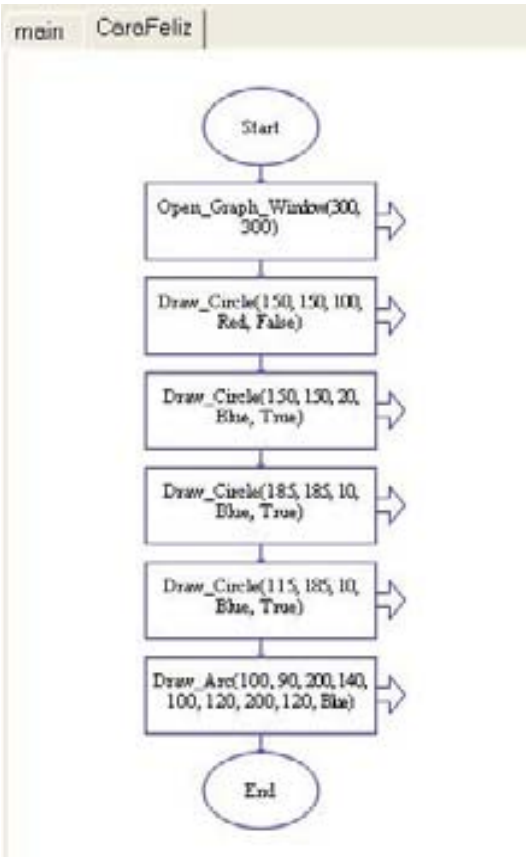
En el quinto **call** colocaremos la siguiente instrucción.

Draw_Circle(115,185,10,Blue,True)

En el sexto **call** colocaremos la siguiente instrucción.

Draw_Arc(100,90,200,140,100,120,200,120,Blue)

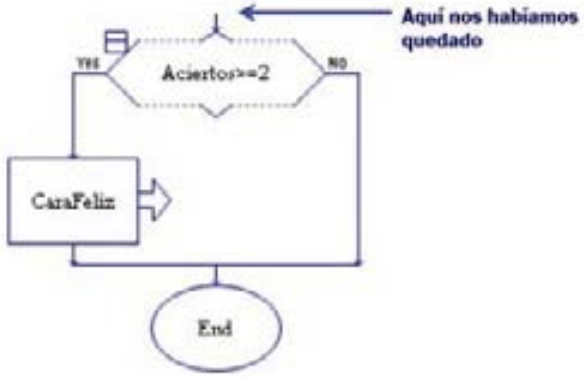
Finalmente nuestro *su c art* quedaría como sigue.



11. Vaya al procedimiento principal del programa (*main*) haciendo clic sobre la pestaña correspondiente en la parte superior en *or space*.

12. Al final del programa agregue un **selection**, y en la salida por verdadero de dicha estructura agregue una instrucción **call**. Como **selection condition** especifique Aciertos>=2, y en el **call** escriba el nombre del *su c art* que creó, es decir, CaraFeliz. El resultado será que, si se tuvieron dos o más aciertos, el programa lanzará una cara feliz que indica que ha sido ganador.

13. La parte final de nuestro programa quedaría como sigue.



14. Pruebe su programa y vea los resultados.

1 . Una forma práctica de probar el programa es repitiendo un mismo número del 1 al 3 las cinco veces en que se pregunta un número. Intente también de forma real, escribiendo los números aleatoriamente. Este programa puede servir como base para crear un juego de lanzamiento de datos, que incluso podrían dibujarse.

16. Juegue un poco. suerte.

1 . Grabe su programa, utilizando cualquiera de las formas de guardado: menú *ile ave*, trl , o haciendo clic en *uardar diagrama* de *ool ar*.

FIN DEL EJERCICIO ✱

Programación de casos prácticos usando Raptor

En esta parte se programarán los casos prácticos utilizando Raptor. Básicamente se toman como referencia las miniespecificaciones. Cabe aclarar que no es necesario haber realizado las miniespecificaciones para realizar los programas en Raptor, dado que puede hacerse directamente a partir de la descripción del problema, a partir del algoritmo y del algoritmo orientado a datos.

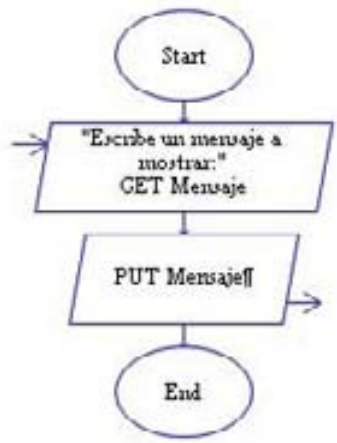
Caso 1: Mensaje

Este programa tiene como finalidad comprobar técnicas y herramientas de desarrollo. Lo que el programa debe hacer es preguntar por un mensaje de saludo, para posteriormente mostrar el mensaje que se le proporcionó.

Miniespecificación

```
INICIO:
        (Mensaje, x(30))
        « "Escribe un mensaje a mostrar: "
        » Mensaje
        « Mensaje
:FIN
```

Programa en Raptor



Caso 2: Niveles de servicio

Se desea saber la disponibilidad del servicio de Internet para una persona, expresado en horas de disponibilidad.

Dicho nivel de servicio depende del grado de disponibilidad que ofrece la computadora que utiliza, así como del grado de disponibilidad del enlace a Internet que utiliza la máquina.

El horario de labores de la persona que utiliza la computadora y el enlace a Internet es de 8 horas.

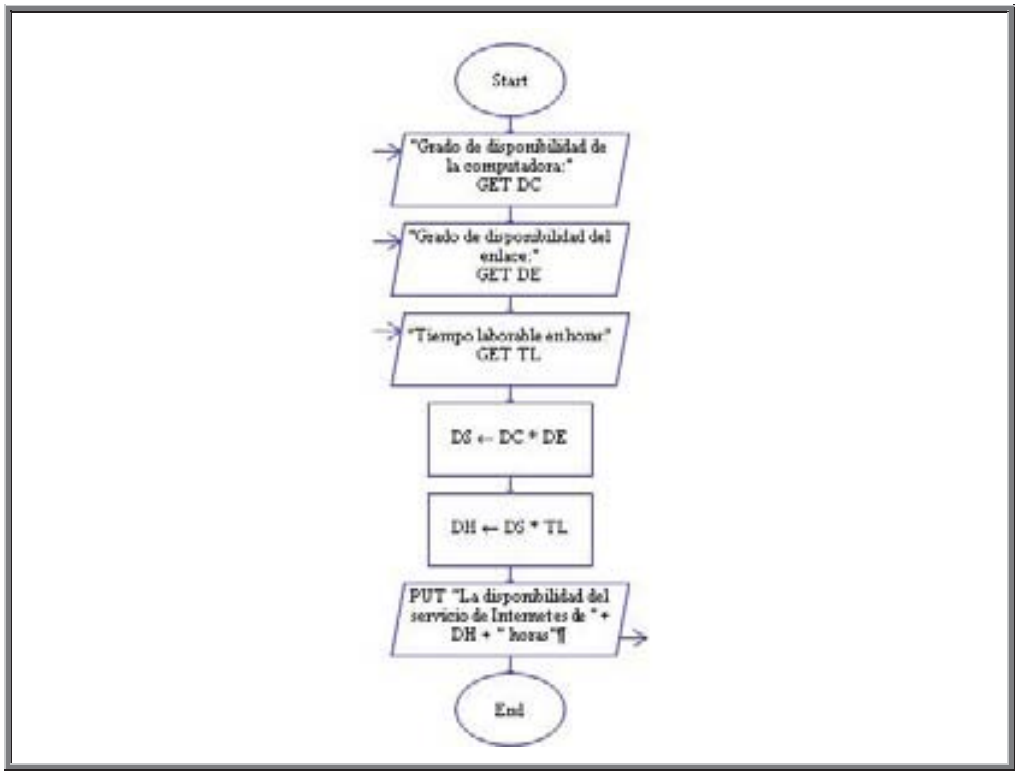
Para poder realizar los cálculos adecuadamente, tome en cuenta que la disponibilidad de un sistema de dispositivos (dos o más dispositivos que trabajan de forma conjunta) se calcula multiplicando los porcentajes de grado de disponibilidad de cada uno de ellos, y el resultado siempre será menor a la disponibilidad del menor.

Para calcular el tiempo de disponibilidad de un servicio (en este caso el acceso a la información publicada en Internet), sólo basta con multiplicar el tiempo en que es requerido el servicio por la disponibilidad del sistema de dispositivos.

Miniespecificación

```
1   INICIO:
2           (DC,d[0-1])
3           (DE,d[0-1])
4           (TL,d[0-1])
5           (DS,d[0-1])
6           (DH,d[0-1])
7           « "Grado de disponibilidad de la computadora:"
8           » DC
9           « "Grado de disponibilidad del enlace:"
10          » DE
11          « "Tiempo laborable en horas:"
12          » TL
13          DS ← DC * DE
14          DH ← DS * TL
15          « "La disponibilidad del servicio de Internet es de " + DH + " horas"
16  :FIN
```

Programa en Raptor



Caso 3: Muestra de una tabla de multiplicar

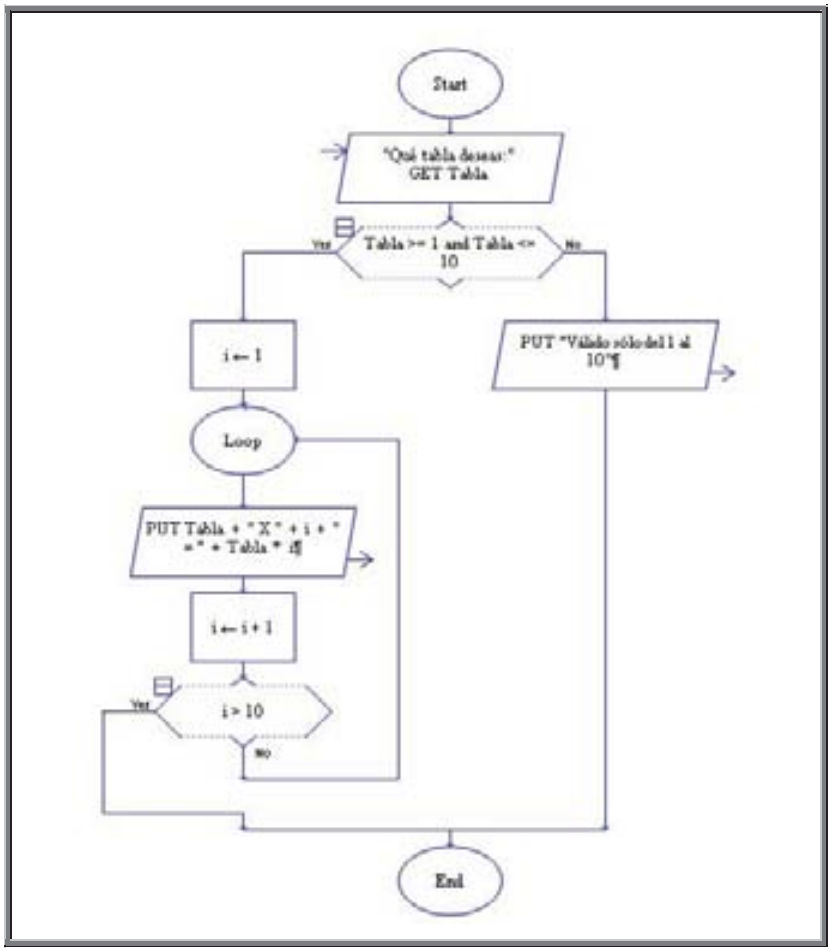
El programa debe mostrar cualquier tabla de multiplicar que le solicitemos. La única restricción es que la tabla deberá ser de un número mayor o igual a 1, y menor o igual a 10.

Si se pide una tabla de un número que no esté entre el 1 y el 10, el programa debe indicar, mediante un mensaje, que no puede continuar.

Miniespecificación (usando estructuras de control)

```
INICIO:
        (i,i[0-n])
        (Tabla,i[1-10])
        « "Qué tabla deseas: "
        » Tabla
        Si Tabla >= 1 ∧ Tabla <=10 Entonces
                i ← 1
                Inicio
                        « Tabla + " X" + i + "=" + Tabla * i
                        i ← i + 1
                Hasta que i > 10
        De lo contrario
                « "Válido sólo del 1 al 10"
        Fin Si
:FIN
```

Programa en Raptor



NOTA

TIP DEL PROGRAMADOR: MANEJO DE TIPOS

Adquiera la costumbre de colocar a las variables que sean de tipos de datos que permiten decimales, una referencia decimal, aunque el número por asignar sea entero.

Medida ← 10	Poco recomendable
Medida ← 10.0	Correcto
Medida ← 0.10	Correcto

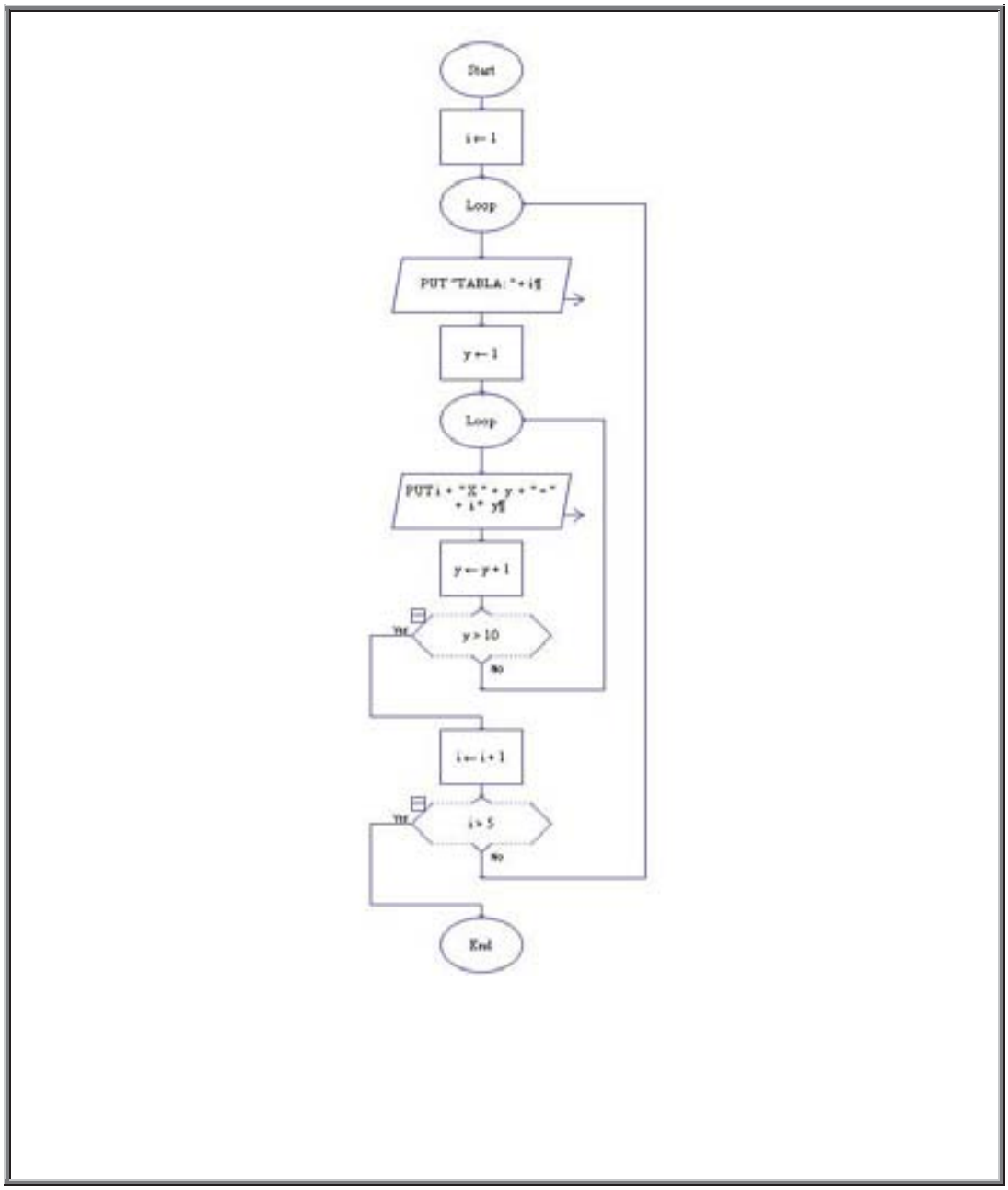
Caso 4: Muestra las tablas de multiplicar del 1 al 5

El programa debe mostrar todas las tablas de multiplicar del 1 al 5, incluyendo un encabezado por tabla.

Miniespecificación

```
INICIO:
        (i,i[0-n])
        (y,i[0-n])
        i ← 1
        Inicio
                « "TABLA: " + i
                y ← 1
                Inicio
                        « i + " X" + y + "=" + i * y
                        y ← y + 1
                Hasta que y > 10
                i ← i + 1
        Hasta que i > 5
:FIN
```

Programa en Raptor



Caso 5: Análisis de promedios

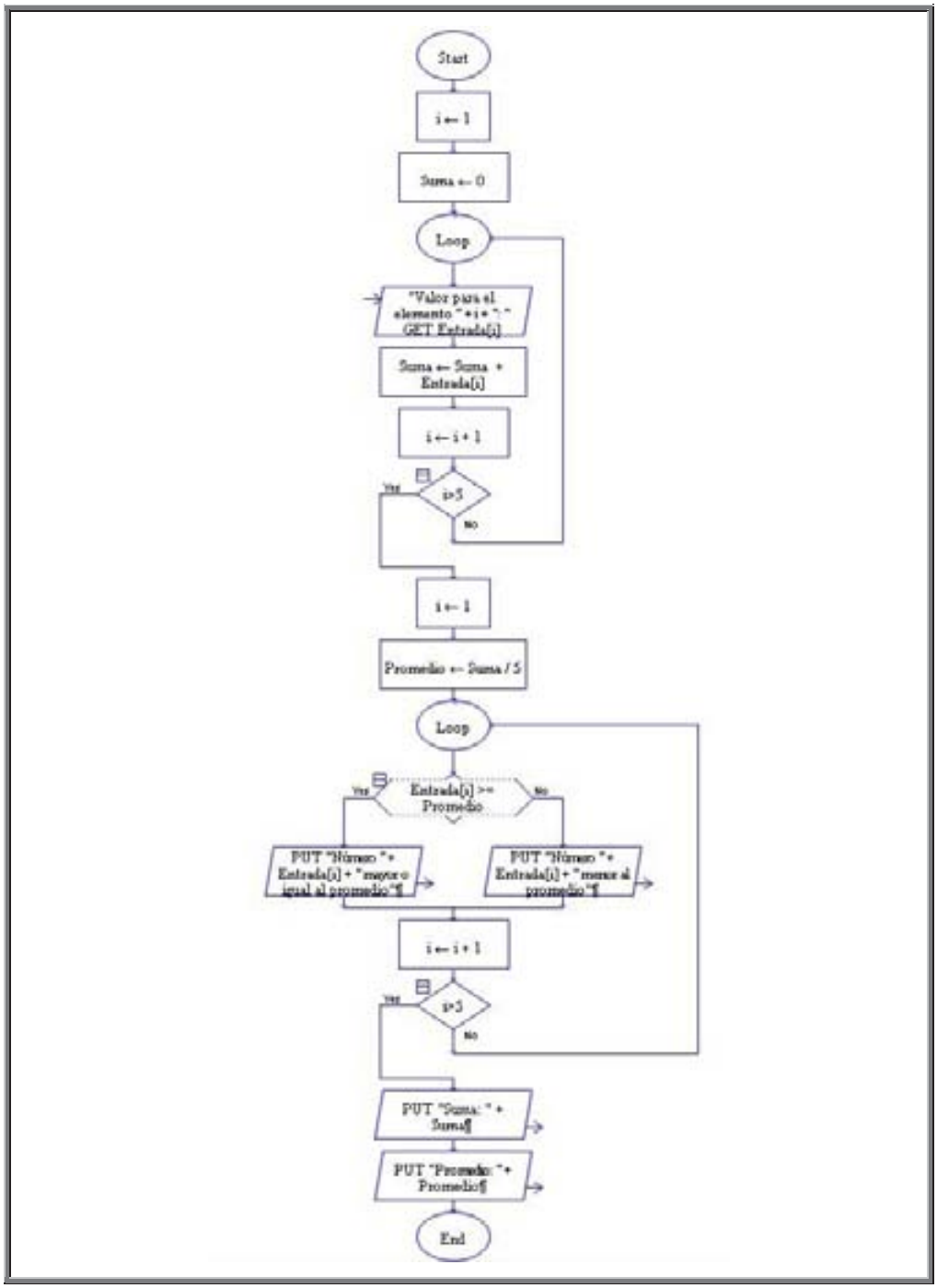
El programa debe pedir 5 números, para después mostrarlos indicando si están o no por encima de su promedio. Al final deberá enviarse un resumen en donde se indique la suma total de los números, y el promedio calculado.

Para la petición de datos utilice un arreglo.

Miniespecificación (usando estructuras)

```
1   INICIO:
2           (i,i[0-n])
3           (Entrada(5),i[0-n])
4           (Suma,d[0-n])
5           (Promedio,d[0-n])
6           Suma ← 0
7           i ← 1
8           Inicio
9                   « “Valor para el elemento ” + i + “: “
10                  » Entrada(i)
11                  Suma ← Suma + Entrada(i)
12                  i ← i + 1
13          Hasta que i > 5
14          Promedio ← Suma / 5
15          i ← 1
16          Inicio
17                  Si Entrada(i) >= Promedio Entonces
18                      « “Número ” + Entrada(i) + “ mayor o igual al promedio: “
19                  De lo contrario
20                      « “Número ” + Entrada(i) + “ menor al promedio: “
21                  Fin Si
22                  i ← i + 1
23          Hasta que i > 5
24          « “Suma: ” + Suma
25          « “Promedio: ” + Promedio
26  :FIN
```

Programa en Raptor



Ejercicio 09.10

Desarrollando un programa de regla de tres simple usando Raptor

Elabore el programa en Raptor para el siguiente caso.

Caso 6: Regla de tres simple

Elabore un programa que realice el cálculo de una regla de tres simple. Como sabe, dicha regla permite calcular la proporción de un número respecto a otro que actúa como referencia de la proporción. Este programa contesta preguntas del tipo ¿qué porcentaje de 200 es 135? El 200 sería el número base de la proporción, mientras que el 135 sería la proporción. La solución se calcula multiplicando la proporción por 100, y dividiéndolo entre el número base de la proporción.

El programa deberá mostrar el resultado del cálculo.

FIN DEL EJERCICIO ✱

NOTA

TIP DEL PROGRAMADOR: LECTURA INVERSA

En ocasiones se requiere un bucle descendente. En donde el contador reduce su valor. Este tipo de bucle se implementa de la siguiente forma,

```
Contador ← Límite superior
Inicio
        Proceso
        Contador = Contador - 1
Hasta  que Contador < LimiteInferior
```

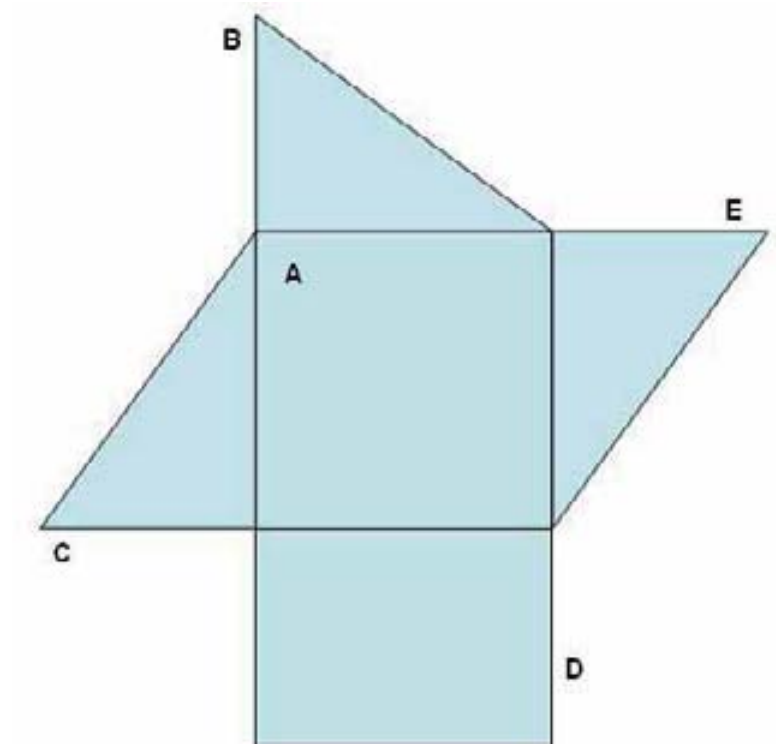
Ejercicio 09.11

Desarrollando un programa para el cálculo del área de un polígono irregular usando Raptor

Elabore el programa en Raptor para el siguiente caso.

Caso 7: Cálculo del área de un polígono irregular

Observe la siguiente figura.



El programa debe calcular el área del polígono. La figura se compone de un cuadrado perfecto (A), tres triángulos rectángulos iguales (B, C, E), cuya base mide lo mismo que uno de los lados del cuadrado, y un rectángulo cuyo ancho mide lo mismo que uno de los lados del cuadrado, mientras que la altura mide lo mismo que la altura de los triángulos. Se pide que haga un programa que calcule el área total.

Por cultura general, sabemos que el área de un rectángulo se calcula multiplicando su base por su altura; el área de un triángulo rectángulo se calcula multiplicando su base por su altura, y luego dividiendo el resultado entre dos; finalmente, el área de un cuadrado se calcula elevando al cuadrado la medida de uno de sus lados.

La parte interesante de este problema es determinar el mínimo número de datos que hay que preguntar.

1. Si cree que es muy bueno, incluya en este ejemplo un subchart que dibuje el polígono en cuestión.

FIN DEL EJERCICIO ✱

Ejercicio 09.12

Desarrollando un programa con bucles, condicionales y arreglos usando Raptor

Elabore el programa en Raptor para el siguiente caso.

Caso 8: Determinando un descuento

Una tienda departamental ofrece descuentos directos sobre las compras que los clientes realizan.

El descuento normal va en función al tipo de cliente que la persona sea: A los clientes normales se les ofrece un 10 de descuento, y a los clientes premier un 15 de descuento.

Actualmente se tiene una promoción: en la compra de 5 artículos, se ofrecerá un descuento adicional al normal. Si el monto de la compra no excede de 1,000 pesos, se agrega un 5 de descuento; si el monto de la compra está entre 1,000 y 2,000 pesos, se agrega un descuento del 10 ; si la compra es mayor a 2,000 pesos, entonces el descuento adicional será del 15 .

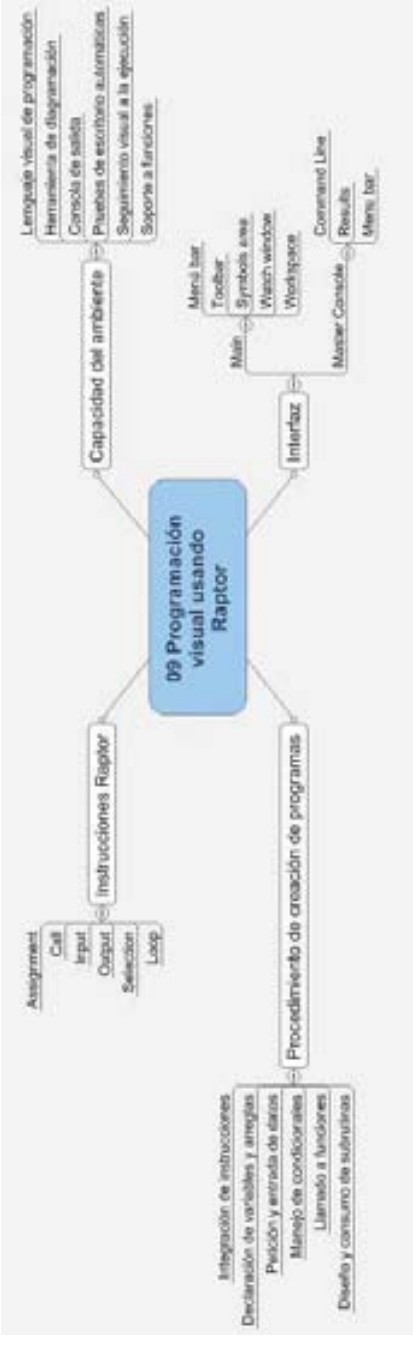
En ese sentido, el menor descuento que se puede obtener es cuando un cliente normal compra menos de 1,000 pesos (15), y el máximo descuento que se puede obtener es cuando un cliente premier compra más de 2,000 pesos (30).

El programa deberá pedir la información de los precios de 5 productos, y determinar el monto de descuento aplicable.

Utilice arreglos.

FIN DEL EJERCICIO ✱

Mapa mental del capítulo



Terminología

Academia de la Fuerza Aérea de los Estados Unidos, 210
Area de resultados, 216
Area de seguimiento, 214
Area de símbolos, 214
Area de trabajo, 214
Assignment, 218
Barra de herramientas, 213
Barra de menú, 213
Barra de menú, 215
Botón de limpiado, 216
Breakpoints, 244
Call, 218
Características de Raptor, 212
Clear, 216
Command line, 216
Condición de selección, 240
Consola maestra, 213
Funciones intrínsecas, 246
Input, 218
Instrucción de asignación, 218
Instrucción de bucle, 218
Instrucción de decisión, 218
Instrucción de entrada, 218
Instrucción de llamado, 218
Instrucción de salida, 218
Línea de comandos, 216
Loop, 218
Main, 213
Master console, 213
Menu bar, 213
Menu bar, 215
Output, 218
Procedimiento principal del programa, 246
Procedimientos externos, 246
Puntos de interrupción, 244
Rapid Algoritmic Prototyping Tool for Ordered Reasoning, 210
Raptor, 210
Results, 216
Salida por falso, 240
Salida por verdadero, 240
Selection condition, 240
Selection, 218
Subcharts, 246
Subdiagramas, 246
Symbols area, 214
Tiempo de diseño, 211
Tiempo de ejecución, 211
Toolbar, 213
United States Air Force Academy, 210
USAFA, 210
Ventana principal, 213
Watch window, 214
Workspace, 214

Preguntas

9.1 ¿Cuál considera que sea la más grande limitación de Raptor como herramienta de desarrollo?

9.2 ¿En qué casos de la vida real podría utilizarse Raptor como herramienta de desarrollo? Cite un ejemplo.

9.3 ¿Cuál fue su experiencia desarrollando en Raptor? ¿Lo seguirá utilizando? Razone su respuesta.

Notas:

Examen rápido

Parte 1 : Marque co n una cruz el inciso que considere que responde a la pregunta.

1) Es la ventana principal de Raptor.
(a) Master Console
(b) Main
(c) Central

2) Es la ventana desde la cual se pueden ejecutar comandos en forma de línea de comandos.
(a) Main
(b) Central
(c) Master Console

3) Es la instrucción raptor para las peticiones de datos.
(a) Output
(b) Input
(c) Selection

4) Es el tipo de bucle permitido por Raptor.
(a) De condición al final
(b) De condición al inicio
(c) Raptor no soporta bucles

5) Es la instrucción que permite incrementar un contador.
(a) Assignment
(b) Process
(c) Input

Parte 2: Coloque en la línea la letra «F» si la sentencia es falsa, y «V» si la sentencia es verdadera.

6) Es posible cometer errores de sintaxis en Raptor.

7) En Raptor no se escriben líneas de código, ya que el programa es en base a símbolos.

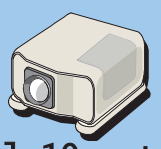
8) Raptor tiene instrucciones básicas.

9) En Raptor no es necesario declarar las variables.

10) En Raptor hay dos formas de agregar una instrucción: desde or area y utilizando el menú contextual.

10

Implementación de algoritmos en Visual Basic.NET



lp10.ppt

Objetivos: Aprenderá a implementar algoritmos utilizando el lenguaje de programación Visual Basic.NET

1. Aprenderá cuáles son los elementos que componen un ambiente de desarrollo.
2. Aprenderá cuáles son los requerimientos de ambiente para desarrollar en Visual Basic.NET
3. Aprenderá las generalidades del lenguaje Visual Basic.NET, así como la estructura básica de sus programas.
4. Aprenderá a realizar cambios de estado, peticiones de datos, muestra de datos, condicionales y bucles usando Visual Basic.NET
5. Aprenderá a compilar y ejecutar programas desarrollados en Visual Basic.NET

www.detodoprogramacion.com

Contenido

Elementos de un ambiente de desarrollo 269
Implementación de algoritmos en Visual Basic.NET 270
Ambiente de desarrollo 270
Estructura básica de un programa 273
Características generales del lenguaje 273
Tipos de datos 274
Declaración de variables y arreglos 274
Operadores principales 275
Cambios de estado 276
Mostrado de datos 277
Peticiones de datos 278
Estructuras de decisión 279
Estructuras de control 280
Compilación y ejecución de programas 282
Codificación de C1Mensaje en Visual Basic.NET 282
Codificación de C2Niveles en Visual Basic.NET 284
Codificación de C3Tabla en Visual Basic.NET 286
Codificación de C4MultiTabla en Visual Basic.NET 288
Codificación de C5Promedios en Visual Basic.NET 291
Desarrollando un programa de regla de tres simple usando Visual Basic.NET 294
Desarrollando un programa para el cálculo del área de un polígono irregular usando Visual Basic.NET 294
Desarrollando un programa con bucles, condicionales y arreglos usando Visual Basic.NET 296
Mapa mental del capítulo 297
Terminología 298
Preguntas 298
Examen rápido 299

En el proceso de comunicación no importa lo ue el emisor diga importa lo ue escucha el receptor.

De nada servirá que tengamos montones de documentos técnicos que nos expliquen lo que debe hacer un programa y la forma en que debe hacer las cosas: a la computadora eso no le representan nada.

Podemos tener claro un algoritmo, un algoritmo orientado a datos, un diagrama de flujo o una miniespecificación detallada, pero para la computadora eso no tiene significado porque no representa secuencias binarias entendidas como órdenes.

Existen herramientas como Raptor que permiten desarrollar de forma visual, evitándonos el trabajo de codificar en un lenguaje determinado. La desventaja es que para ejecutar los programas en Raptor requerimos tener instalada la herramienta, cosa que no será posible en todos los escenarios. Además, si queremos desarrollar actividades más complejas tales como el acceso a bases de datos,

desarrollo eb, manipulación de puertos y dispositivos, etcétera, requeriremos sin duda de lenguajes bastante más poderosos.

En resumen, es inevitable que aprendamos a codificar programas utilizando lenguajes de programación. La buena noticia es que teniendo realizado el análisis y documentados los procesos, pasar a la codificación es una tarea muy simple.

En este libro cubriremos los lenguajes con mayor difusión (*mar et share*) del mercado: Visual Basic.NET, C , C y Java.

NOTA

Como no se trata de un libro de lenguajes de programación, sino uno de lógica de programación, se tratarán los temas del lenguaje en términos mínimos, suficientes y necesarios para la implementación de los casos desarrollados en el texto.

Teniendo ya las miniespecificaciones sólo será necesario hacer unas precisiones en cuanto a los ambientes de desarrollo, particularidades de la sintaxis, palabras reservadas y estructuras.

A partir de aquí cubriremos el último paso que nos falta para ser desarrolladores completos: traducir nuestros algoritmos en código máquina, finalmente.

Elementos de un ambiente de desarrollo

Un *ambiente de desarrollo* es el conjunto de programas, herramientas y configuraciones que permiten la codificación, compilación y ejecución de un programa en un lenguaje de programación dado.

Generalmente en un ambiente de programación nos podemos encontrar los siguientes elementos.

- *arco de trabajo*.

  El *marco de trabajo* es el conjunto de programas y configuraciones requeridos por un ambiente de trabajo para la coordinación del manejo de dispositivos en tiempo de ejecución. El marco de trabajo mantiene el contacto entre los programas ejecutables de un lenguaje, el sistema operativo, y la biblioteca de clases del lenguaje. Si el marco de trabajo no está instalado, los programas no se podrán ejecutar.

- *Editor de programas*.

  El *editor de programas* es el programa que permite la edición del código fuente, proporcionando ayuda para mantener el código fuente estructurado de una forma ordenada y limpia. Dependiendo la herramienta, podrá contar o no con asistentes y facilidades tales como ejecución integrada de programas, puntos de interrupción, manejo avanzado de clipboard, ayuda

interactiva del uso del lenguaje, diferenciación de elementos de código basado en colores, etcétera.

Los editores especializados de algunos lenguajes de programación son muy costosos, pero ahorran una cantidad muy importante de tiempo de desarrollo. A fin de cuentas, dado que los programas terminan siendo archivos de texto plano, se puede utilizar el bloc de notas o el programa Edit.exe de la línea de comandos, para escribir el código fuente.

- *ompilador de programas*.

  El compilador es el programa que se encarga de transformar el código fuente en código intermedio o código ejecutable.

- *iblioteca de clases*.

  Una *biblioteca de clases* es un repositorio de funcionalidad disponible para el lenguaje, que puede ser consumida de manera nativa desde los programas. Casi todos los lenguajes de propósito general poseen clases para el manejo de datos, gráficos, matemáticas, seguridad, elementos de interfaces gráficas, etcétera.

  Sin las bibliotecas de clases el programador tendría que codificar muchas tareas comunes, y perdería mucho tiempo. Gracias a las bibliotecas de clases, el programador se tiene que preocupar sólo por codificar aquello que forma parte de las reglas del negocio, y no cuestiones generales que ya están preconstruidas y disponibles.

Implementación de algoritmos en Visual Basic.NET

Visual Basic.NET es un lenguaje de propósito general orientado a objetos, que permite desarrollar aplicaciones bajo la plataforma de desarrollo de .NET Frame or .

Ambiente de desarrollo

El ambiente de desarrollo del lenguaje Visual Basic.NET se compone como sigue.

- *arco de trabajo*.

  Para el desarrollo de Visual Basic.NET es necesario el marco de trabajo denominado Microsoft .NET Frame or , disponible desde la página eb de Microsoft Corporation. Para labores de desarrollo, es necesario instalar Microsoft.NET Frame or Redistributable Pac age, y luego Microsoft.NET Frame or SDK. Deberá instalarlos para poder probar los programas en Visual Basic.NET.

  Para tener acceso pleno a las herramientas de desarrollo se recomienda que se agregue a la variable de ambiente PATH, la ruta de las herramientas de desarrollo de .NET Frame or .

Las pruebas de este libro se hicieron con la versión 2.0 de .NET Frame or SDK, que puede descargarse de forma gratuita en www.microsoft.com. Si es su caso, la ruta de herramientas del marco de trabajo puede agregarse desde línea de comandos, ejecutando lo siguiente:

```
PATH=”%PATH%”;”C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727”
```

Puede variar dependiendo la versión instalada de .NET Frame or (v1.0.3705, v1.1.4322, v2.0.50215) o del sistema operativo (WINNT).

- *Editor de programas.*

  Aunque se pueden editar programas Visual Basic.NET en cualquier editor de texto plano, la herramienta más sofisticada para la programación en dicho lenguaje es Microsoft Visual Studio.NET. Otras opciones que se tienen es Macromedia Dream eaver, Sharp Develop, entre otros.

- *ompilador de programas.*

  El compilador de programas en Visual Basic.NET es el programa vbc.exe, que viene incluido en Microsoft.NET Frame or SDK. Este compilador produce archivos de extensión EXE, que a pesar de la extensión, no son ejecutables puros. Los archivos EXE generados por el compilador son código intermedio, a los que se les da el nombre de *ensamblados*. Al ejecutarse por primera vez los programas EXE desarrollados en Visual Basic.NET, un elemento del .NET Frame or denominado *ust n ime ompiler* realizará la fase de síntesis del proceso de compilación, creando finalmente el código máquina.

- *iblioteca de clases.*

  La biblioteca de clases de Microsoft .NET Frame or está formada por más de 6,000 clases que permiten prácticamente cualquier cosa. Esta biblioteca recibe el nombre de *iblioteca de clases base* (*ase lass ibrary*), que se compone por un conjunto de archivos DLL utilizables por todos los lenguajes .NET.

  En el caso de Visual Basic.NET la biblioteca de clases se instala y está disponible en el equipo al instalar .NET Frame or , por lo que no es necesario hacer referencia a los archivos físicos DLL que contienen las clases.

  Las clases contenidas en los DLL se organizan a través de una jerarquía de clases. Para tener acceso a la funcionalidad de una clase desde un programa es necesario especificar toda su jerarquía al momento de invocarla. Por ejemplo, si deseamos invocar las clases para el manejo de entrada y salida por consola, es necesario invocarlas de la siguiente manera: System.Console.Read() y System.Console.Write(). System es la jerarquía, Console es la clase, y Read() y Write() son los métodos que nos dan realmente la funcionalidad de escritura y lectura.

Como es un poco molesto tener que estar repitiendo como prefijo la jerarquía de clase (System) podemos utilizar un espacio de nombre al inicio el programa, para obviar la referencia a la jerarquía de clases e incluso el nombre de la clase. Un *espacio de nombres* es un medio para la referencia lógica de la funcionalidad contenida en una biblioteca de clases, permitiendo el acceso simplificado a las clases. En Visual Basic.NET podemos establecer espacios de nombre escribiendo al inicio del programa líneas similares a estas.

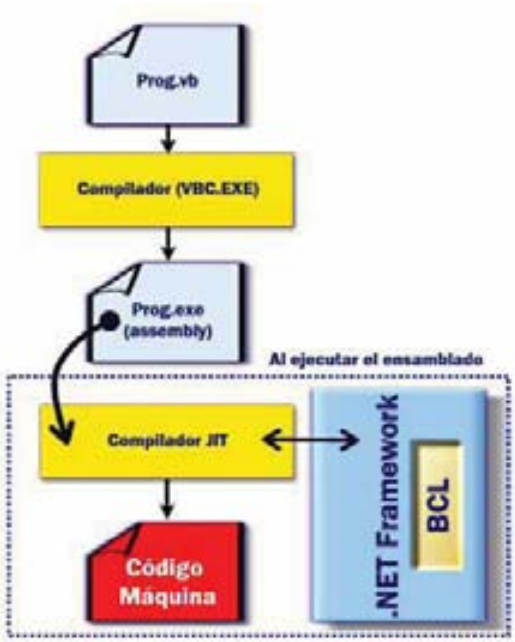
```
Imports System
Imports System.Console
```

Al aplicar las instrucciones anteriores, por ejemplo, los métodos System.Console.Read() y System.Console.Write() podrían referirse simplemente como Write() y Read().

El uso de espacios de nombres nos permite obviar la especificación de la jerarquía de clases al momento de invocar la funcionalidad de las bibliotecas.

FIGURA 10.01

Proceso de compilación y ejecución en Visual Basic.NET



Estructura básica de un programa

La estructura base de todo programa en el lenguaje Visual Basic.NET es la siguiente.

```
1   Imports System
2   Imports System.Console
3
4   Module   ombrePrograma
5      Sub Main()
6          ...
7
8      End Sub
    End Module
```

Por convencionalismo, se debe colocar como nombre del módulo el nombre del programa. Nuestro procedimiento principal deberá incluirse en el espacio que ocupan la línea 6 de la ilustración.

Características generales del lenguaje

A Visual Basic.NET le aplican las siguientes reglas generales básicas.

1. No es sensible a mayúsculas y minúsculas (*non case sensitive*), por lo que es irrelevante el uso de ellas para el compilador.
2. Las estructuras del programa, tales como Module y Sub siempre deben finalizarse con instrucciones End (End Module, End Sub).
3. Las líneas en Visual Basic.NET se terminan al encontrar un salto de línea (CR LF).
4. El continuador de línea en Visual Basic.NET es una línea baja ( ). Antes de un continuador de línea debe haber un espacio en blanco, y después de colocar el continuador no debe haber nada.

. Los comentarios en Visual Basic.NET se especifican colocando un apóstrofe ( ) al inicio de la línea.

6. Para utilizar variables, antes deben declararse.

. El operador & en Visual Basic.NET sirve para concatenar cadenas.

Tipos de datos

Los tipos de datos básicos del lenguaje Visual Basic.NET son los siguientes.

Tipo	Consumo en Bytes	Dominio de tipo
Boolean		r e o alse
Byte	1 byte	0 a 255 (sin signo).
Decimal	16 bytes	0 a 79.228.162.514.264.337.593.543.950.335 sin separador decimal; 0 a 7,9228162514264337593543950335 con 28 posiciones a la derecha del signo decimal; el número más pequeño distinto de cero es 0,0000000000000000000000000001 (1E 28).
Integer	4 bytes	2.147.483.648 a 2.147.483.647

Declaración de variables y arreglos

La forma de declarar variables en el lenguaje Visual Basic.NET es a través de la instrucción Dim. La sintaxis es la siguiente.

```
Dim ariable As ipo [ = E presión ]
```

Donde *ariable* es un nombre de variable válido, y *ipo* es un tipo de datos reconocido por el lenguaje. En Visual Basic.NET se puede asignar un valor de inicio al momento de declarar una variable, asignando una *E presión*, con posterioridad al tipo de dato.

A continuación unos ejemplos.

```
Dim i As Integer
Dim i As Integer = 1
Dim Mensaje As String
```

Equivalencias

Miniespecificación	Diagrama de flujo	Visual Basic.NET
(i,i[1-n])	i = 1	Dim i As Integer = 1
(Suma,d[1-n])	Suma = 0	Dim Suma As Decimal = 0

Los arreglos también se definen en el lenguaje Visual Basic.NET utilizando Dim. La sintaxis es la siguiente.

```
Dim  ariable(  imo  ub ndice) As  ipo
```

Donde *ariable* es un nombre de variable válido, *ipo* es un tipo de datos reconocido por el lenguaje, y *imo ub ndice* es el número del subíndice más alto. En Visual Basic.NET los arreglos son de base cero, es decir, que los subíndices son 0, 1, 2, 3, y así. Declarar un arreglo como Numeros(3), genera un arreglo de 4 elementos 0,1,2,3 . En otras palabras, el número de elementos es *imo ub ndice* 1.

A continuación unos ejemplos.

```
Dim Ventas(2) As Integer
Dim Ventas() As Integer = {20000,15000}
```

Operadores principales

Los operadores principales del lenguaje Visual Basic.NET, aplicables a los ejercicios tratados en el libro, son los siguientes.

Tipo	Operador	Nombre
Asignación	=	Igual
Manejo de cadenas	&	Concatenación
Aritmético	+	Suma
	-	Resta
	*	Multiplicación

	/	División
	^	Exponenciación
Lógico	And	Disyuntiva
	Or	Conjuntiva
	Not	Negación
Comparativos	>	Mayor que
	<	Menor que
	>=	Mayor o igual que
	<=	Menor o igual que
	=	Igual a
	<>	Distinto a
Asignación incluyente	+=	Suma incluyente
	-=	Resta incluyente

Cambios de estado

Los cambios de estado en el lenguaje Visual Basic.NET se realizan utilizando el operador de asignación (=).

ariable = E presión

Donde *ariable* es el nombre de una variable previamente declarada y *E presión* es la expresión que se le asigna a la variable. Estos son unos ejemplos.

```
AreaCirculo = Lado^2
i = i + 1
i += 1
```

En el ejemplo, las últimas dos líneas hacen lo mismo.

Equivalencias

Miniespecificación	Diagrama de flujo	Visual Basic.NET
Promedio ← Suma/5	Promedio = Suma / 5	Promedio = Suma/5
Suma ← Suma + Entrada(i)	Suma = Suma + Entrada(i)	Suma += Entrada(i)
i ← i + 1	i = i + 1	i += 1

Mostrado de datos

El mostrado de datos se realiza en el lenguaje de Visual Basic.NET a través de las órdenes Write y WriteLine. La terminación "Line" de los comandos aplica si deseamos que al final de la instrucción se incluya un salto de línea.

```
Write("E presión")
```

Donde *E presión* es la expresión a mostrar.

Ejemplo.

```
Write("Dame un número: ")
```

Equivalencias

Miniespecificación
« "Suma: " + Suma
Diagrama de flujo
PUT "Suma: " + Suma
Visual Basic.NET
WriteLine("Suma:"& Suma)

Peticiones de datos

Las peticiones de datos se realizan en el lenguaje de Visual Basic.NET a través de las órdenes Read y ReadLine.

Read sirve para leer los datos. La terminación "Line" de los comandos aplica si deseamos que al final de la instrucción se incluya un salto de línea.

```
ariable = ReadLine()
```

Donde *ariable* es la variable en la que se almacenará lo que el usuario introduzca a la interfaz.

Ejemplo.

```
Numero = ReadLine()
```

Generalmente las peticiones de datos se realizan a solicitud, por lo que es común asociar una petición con un mostrado de datos previo, quedando como sigue.

```
Write("E presión")
ariable = ReadLine()
```

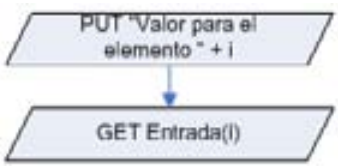
Equivalencias

Miniespecificación

« "Valor para el elemento " + i

» Entrada(i)

Diagrama de flujo



Visual Basic.NET

```
Write("Valor para el elemento " & i)
Entrada(i) = ReadLine()
```

Estructuras de decisión

Las estructuras de decisión se realizan en el lenguaje Visual Basic.NET a través de la estructura If Then Else.

```
If   ondición Then
          nstrucciones por verdadero
[ Else
          nstrucciones por falso ]
End If
```

Donde *ondición* es la expresión lógica a evaluar. En caso de que *ondición* sea True, se ejecutará el bloque de código que contiene las *nstrucciones por verdadero*. De manera opcional se puede especificar un bloque de código a ejecutar si *ondición* resuelve por falso, utilizando Else. En ese caso, se ejecutarían las *nstrucciones por falso*. La estructura If siempre debe terminar (End If).

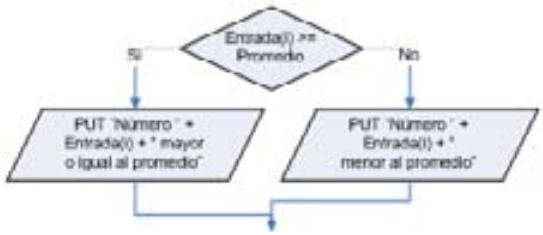
Ejemplos.

```
If Monto > LimiteCredito Then
        Venta = "Rechazada"
Else
        Venta = "Aprobada"
End If
```

Equivalencias

Miniespecificación
Si Entrada(i) >= Promedio Entonces
« "Numero " + Entrada(i) + " mayor o igual al promedio"
De lo contrario
« "Numero " + Entrada(i) + " menor al promedio"
Fin Si

Diagrama de flujo



Visual Basic.NET

```
If  Entrada(i) >= Promedio Then
        WriteLine("Numero " & Entrada(i) & " mayor o igual al promedio")
Else
        WriteLine("Numero " & Entrada(i) & " menor al promedio")
End If
```

Estructuras de control

Las estructuras de control con condición al final se realizan en el lenguaje Visual Basic.NET a través de la estructura Do Loop.

```
Do
        nstrucciones
Loop Until   ondición
```

Donde *nstrucciones* es el bloque de código que se estará repitiendo hasta que *ondición* se cumpla. Todo lo que esté entre Do y Loop se considera código controlado. Es importante no olvidar colocar y cambiar el estado de la variable de trabajo, dentro del código de control.

Ejemplos.

```
i = 1
Do
        WriteLine("Valor del contador: " & i)
        i += 1
Loop Until i > 5
```

Equivalencias

Miniespecificación

i ← 1

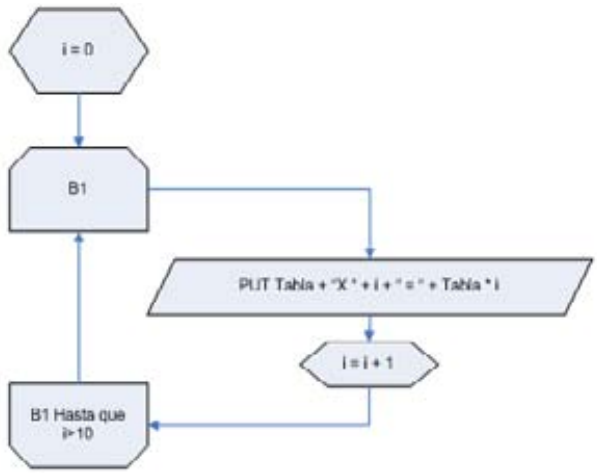
Inicio

« Tabla + “X” + i + “=” + Tabla*i

i ← i + 1

Hasta que i > 10

Diagrama de flujo



Visual Basic.NET

```
i = 1
Do
        WriteLine(Tabla & “X” & i & “=” & Tabla*i)
        i += 1
Loop Until i > 10
```

Aunque no manejamos estructuras de control con condición al inicio en este libro, proporcionamos la sintaxis de su implementación en Visual Basic.NET

```
Do While  ondición
        nstrucciones
Loop
```

Donde *nstrucciones* es el bloque de código que se estará repitiendo mientras que *ondición* se cumpla por verdadero.

Compilación y ejecución de programas

La compilación de los programas se realiza utilizando el compilador de línea de comandos de Visual Basic.NET, llamado VBC.EXE. La sintaxis básica de compilación es la siguiente.

```
vbc Programa.vb
```

Donde *Programa* es el nombre del archivo que desea compilar.

El comando se ejecuta desde la línea de comandos. El programa llamado Fuente.vb es el nombre físico del archivo a compilar. Es importante no omitir la extensión de los programas desarrollados en Visual Basic.NET, que es VB.

Al concluir la compilación sin errores, se habrá creado un programa de extensión .exe, que se trata de *código intermedio de . E* , llamado también (*icrosoft ntermediate anguage*), más conocido aún como *ensamblado* (*assembly*); sólo se podrá ejecutar el ensamblado en equipos con .NET Frame or instalado.

El programa podrá ejecutarse desde la línea de comandos, invocando el programa ejecutable que se generó.

```
Programa
```

Donde Programa es el nombre del archivo .exe que desea ejecutar.

Al ejecutar el código ensamblado, el Just In Time Compiler de .NET Frame or detecta que no se trata de código máquina, realiza la fase de síntesis del proceso de compilación a partir del código intermedio de .NET, generando el código máquina optimizado para el equipo desde el cual se intentó la ejecución, mostrando finalmente los resultados del programa.

Ejercicio 10.01

Codificación de C1Mensaje en Visual Basic.NET

En el presente ejercicio se codificará el primer caso de estudio, utilizando el lenguaje Visual Basic.NET.

1. Utilice un editor de texto plano (*loc de notas*, por ejemplo), para escribir el código fuente llamado C1Mensaje.vb; el archivo deberá guardarse en el directorio de trabajo del libro (C:\APIP).

Este programa tiene como finalidad comprobar técnicas y herramientas de desarrollo. Lo que el programa debe hacer es preguntar por un mensaje de saludo, para posteriormente mostrar el mensaje que se le proporcionó.

Miniespecificación

```
1   INICIO:
2           (Mensaje, x(30))
3           « "Escribe un mensaje a mostrar: "
4           » Mensaje
5           « Mensaje
6   :FIN
```

Codificación de **C1Mensaje.vb**

```
1   Imports System
2   Imports System.Console
3
4   Module C1Mensaje
5       Sub Main()
6
7           Dim Mensaje As String
8           Write("Escribe un mensaje a mostrar: ")
9           Mensaje = ReadLine()
10          WriteLine(Mensaje)
11
12      End Sub
13  End Module
```

2. Compile el programa utilizando la siguiente línea.

```
vbc C1Mensaje.vb
```

3. Si no genera errores, ejecute el programa de la siguiente forma.

```
C1Mensaje.exe
```

Si todo ocurrió como se esperaba, la salida será la siguiente:

Entrada	Escribe un mensaje a mostrar: Aprenda
Salida	Aprenda

FIN DEL EJERCICIO ✱

Ejercicio 10.02

Codificación de C2Niveles en Visual Basic.NET

En el presente ejercicio se codificará el segundo caso de estudio, utilizando el lenguaje Visual Basic.NET.

1. Utilice un editor de texto plano (*loc de notas*, por ejemplo), para escribir el código fuente llamado C2Niveles.vb; el archivo deberá guardarse en el directorio de trabajo del libro (C:\APIP).

Se desea saber la disponibilidad del servicio de Internet para una persona, expresado en horas de disponibilidad.

Dicho nivel de servicio depende del grado de disponibilidad que ofrece la computadora que utiliza, así como del grado de disponibilidad del enlace a Internet que utiliza la máquina.

El horario de labores de la persona que utiliza la computadora y el enlace a Internet es de 8 horas.

Para poder realizar los cálculos adecuadamente, tome en cuenta que la disponibilidad de un sistema de dispositivos (dos o más dispositivos que trabajan de forma conjunta) se calcula multiplicando los porcentajes de grado de disponibilidad de cada uno de ellos, y el resultado siempre será menor a la disponibilidad del menor.

Para calcular el tiempo de disponibilidad de un servicio (en este caso el acceso a la información publicada en Internet), sólo basta con multiplicar el tiempo en que es requerido el servicio por la disponibilidad del sistema de dispositivos.

www.detodoprogramacion.com

Miniespecificación

```
1   INICIO:
2           (DC,d[0-1])
3           (DE,d[0-1])
4           (TL,d[0-1])
5           (DS,d[0-1])
6           (DH,d[0-1])
7           « "Grado de disponibilidad de la computadora:"
8           » DC
9           « "Grado de disponibilidad del enlace:"
10          » DE
11          « "Tiempo laborable en horas:"
12          » TL
13          DS ← DC * DE
14          DH ← DS * TL
15          « "La disponibilidad del servicio de Internet es de " + DH + " horas"
16  :FIN
```

Codificación de **C2Niveles.vb**

```
1   Imports System
2   Imports System.Console
3
4   Module Niveles
5       Sub Main()
6
7           Dim DC As Decimal
8           Dim DE As Decimal
9           Dim TL As Decimal
10          Dim DS As Decimal
11          Dim DH As Decimal
12
13          Write("Grado de disponibilidad de la computadora: ")
14          DC = ReadLine()
15          Write("Grado de disponibilidad del enlace: ")
16          DE = ReadLine()
17          Write("Tiempo laborable en horas: ")
18          TL = ReadLine()
19
20          DS= DC*DE
21          DH= Ds*TL
22
23          WriteLine("La disponibilidad del servicio de Internet es de " & DH &" horas")
24
25      End Sub
26  End Module
```

2. Compile el programa utilizando la siguiente línea.

```
vbc C2Niveles.vb
```

3. Si no genera errores, ejecute el programa de la siguiente forma.

```
C2Niveles.exe
```

Si todo ocurrió como se esperaba, la salida será la siguiente:

Entrada	Grado de disponibilidad de la computadora: .98
Entrada	Grado de disponibilidad del enlace: .85
Entrada	Tiempo laborable en horas: 8
Salida	La disponibilidad del servicio de internet es de 6.6640 horas

FIN DEL EJERCICIO ✱

Ejercicio 10.03

Codificación de C3Tabla en Visual Basic.NET

En el presente ejercicio se codificará el tercer caso de estudio, utilizando el lenguaje Visual Basic.NET.

1. Utilice un editor de texto plano (*loc de notas*, por ejemplo), para escribir el código fuente llamado C3Tabla.vb; el archivo deberá guardarse en el directorio de trabajo del libro (C:\APIP).

El programa debe mostrar cualquier tabla de multiplicar que le solicitemos. La única restricción es que la tabla deberá ser de un número mayor o igual a 1, y menor o igual a 10.

Si se pide una tabla de un número que no esté entre el 1 y el 10, el programa debe indicar, mediante un mensaje, que no puede continuar.

Miniespecificación

```
1   INICIO:
2           (i,i[0-n])
```

```
3          (Tabla,i[1-10])
4          « "Qué tabla deseas: "
5          » Tabla
6          Si Tabla >= 1 ∧ Tabla <=10 Entonces
7                  i ← 1
8                  Inicio
9                          « Tabla + " X" + i + "=" + Tabla * i
10                         i ← i + 1
11                 Hasta que i > 10
12         De lo contrario
13                 « "Válido sólo del 1 al 10"
14         Fin Si
15 :FIN
```

Codificación de **C3Tabla.vb**

```
1  Imports System
2  Imports System.Console
3
4  Module C3Tabla
5     Sub Main()
6
7        Dim Tabla As Integer
8        Dim i As Integer
9
10       Write("Qué tabla deseas: ")
11       Tabla = ReadLine()
12
13       If Tabla >=1 And Tabla <= 10 Then
14          i=1
15          Do
16             WriteLine(Tabla & " X " & i & " = "& Tabla*i )
17             i += 1
18          Loop Until i > 10
19       Else
20          WriteLine("Válido sólo del 1 al 10")
21       End If
22
23    End Sub
24 End Module
```

2. Compile el programa utilizando la siguiente línea.

```
vbc C3Tabla.vb
```

3. Si no genera errores, ejecute el programa de la siguiente forma.

```
C3Tabla.exe
```

Si todo ocurrió como se esperaba, la salida será la siguiente:

Entrada	Que tabla deseas: 5
Salida	5 X 1 = 5
	5 X 2 = 10
	5 X 3 = 15
	5 X 4 = 20
	5 X 5 = 25
	5 X 6 = 30
	5 X 7 = 35
	5 X 8 = 40
	5 X 9 = 45
	5 X 10 = 50

Entrada	Que tabla deseas: 25
Salida	Válido sólo del 1 al 10

FIN DEL EJERCICIO ✱

Ejercicio 10.04

Codificación de C4MultiTabla en Visual Basic.NET

En el presente ejercicio se codificará el cuarto caso de estudio, utilizando el lenguaje Visual Basic.NET.

1. Utilice un editor de texto plano (*loc de notas*, por ejemplo), para escribir el código fuente llamado C4MultiTabla.vb; el archivo deberá guardarse en el directorio de trabajo del libro (C:\APIP).

El programa debe mostrar todas las tablas de multiplicar del 1 al 5, incluyendo un encabezado por tabla.

Miniespecificación

```
1   INICIO:
2           (i,i[0-n])
3           (y,i[0-n])
4           i ← 1
5           Inicio
6                   « "TABLA: " + i
7                   y ← 1
8                   Inicio
9                           « i + " X" + y + "=" + i * y
10                          y ← y + 1
11                  Hasta que y > 10
12                  i ← i + 1
13          Hasta que i > 5
14  :FIN
```

Codificación de **C4MultiTabla.vb**

```
1   Imports System
2   Imports System.Console
3
4   Module C4MultiTabla
5       Sub Main()
6
7           Dim i As Integer
8           Dim y As Integer
9
10          i=1
11          Do
12              WriteLine("Tabla: " & i)
13              y=1
14              Do
15                  WriteLine(i & " X " & y & " = "& i*y )
16                  y += 1
17              Loop Until y > 10
18              i += 1
19          Loop Until i > 5
20
21      End Sub
22  End Module
```

2. Compile el programa utilizando la siguiente línea.

```
vbc C4MultiTabla.vb
```

3. Si no genera errores, ejecute el programa de la siguiente forma.

```
C4MultiTabla.exe
```

Si todo ocurrió como se esperaba, la salida será la siguiente:

Salida

```
Tabla: 1
1 X 1 = 1
1 X 2 = 2
1 X 3 = 3
1 X 4 = 4
1 X 5 = 5
1 X 6 = 6
1 X 7 = 7
1 X 8 = 8
1 X 9 = 9
1 X 10 = 10
Tabla: 2
2 X 1 = 2
2 X 2 = 4
2 X 3 = 6
2 X 4 = 8
...

5 X 7 = 35
5 X 8 = 40
5 X 9 = 45
5 X 10 = 50
```

FIN DEL EJERCICIO ✱

Ejercicio 10.05

Codificación de C5Promedios en Visual Basic.NET

En el presente ejercicio se codificará el quinto caso de estudio, utilizando el lenguaje Visual Basic.NET.

1. Utilice un editor de texto plano (*loc de notas*, por ejemplo), para escribir el código fuente llamado C5Promedios.vb; el archivo deberá guardarse en el directorio de trabajo del libro (C:\APIP).

El programa debe pedir 5 números, para después mostrarlos indicando si están o no por encima de su promedio. Al final deberá enviarse un resumen en donde se indique la suma total de los números, y el promedio calculado.

Miniespecificación

```
1   INICIO:
2           (i,i[0-n])
3           (Entrada(5),i[0-n])
4           (Suma,d[0-n])
5           (Promedio,d[0-n])
6           Suma ← 0
7           i ← 1
8           Inicio
9                   « "Valor para el elemento " + i + ": "
10                  » Entrada(i)
11                  Suma ← Suma + Entrada(i)
12                  i ← i + 1
13          Hasta que i > 5
14          Promedio ← Suma / 5
15          i ← 1
16          Inicio
17                  Si Entrada(i) >= Promedio Entonces
18                      « "Número " + Entrada(i) + " mayor o igual al promedio: "
19                  De lo contrario
20                      « "Número " + Entrada(i) + " menor al promedio: "
21                  Fin Si
22                  i ← i + 1
23          Hasta que i > 5
24          « "Suma: " + Suma
25          « "Promedio: " + Promedio
26  :FIN
```

Codificación de **C5Promedios.vb**

```
1   Imports System
2   Imports System.Console
3
4   Module C5Promedios
5      Sub Main()
6
7         Dim i as Integer
8         Dim Entrada(4)As Integer
9         Dim Suma as Integer
10        Dim Promedio As Decimal
11        Dim Valor as Integer
12
13        Suma=0
14
15        i=1
16        Do
17           Write("Valor para el elemento " & i & ": ")
18           Entrada(i-1)=ReadLine()
19           Suma+=Entrada(i-1)
20           i += 1
21        Loop Until i > 5
22
23        Promedio= Suma/5
24
25        i=1
26        Do
27           If Entrada(i-1)>= Promedio Then
28              WriteLine("Número " & Entrada(i-1) & " mayor o igual al promedio.")
29           Else
30              WriteLine("Número " & Entrada(i-1) & " menor al promedio.")
31           End If
32           i += 1
33        Loop Until i > 5
34
35        WriteLine("Suma: " & Suma)
36        WriteLine("Promedio: " & Promedio )
37
38     End Sub
39  End Module
```

2. Es importante mencionar que este programa tiene discrepancias con la miniespecificación. Mientras que la miniespecificación refiere a los elementos del arreglo como si fueran base 1 (1,2,3,4,5) no debemos olvidar que Visual Basic.NET trabaja los arreglos en base cero (0,1,2,3,4). Esto se manifiesta en dos momentos dentro del programa:

a Al declarar el arreglo, la miniespecificación declara 5 elementos (línea 3), mientras que el programa declara 4 (línea 7); a fin de cuentas es lo mismo, dado que en Visual Basic.NET el 4 significa el valor más grande de subíndice, lo que nos lleva a un número de elementos n 1, es decir, los mismos 5 que señala la miniespecificación.

b En las llamadas al arreglo, en la miniespecificación se coloca como subíndice i (líneas 11, 17, 18 y 20), mientras que en el programa se refiere al subíndice como i-1 (líneas 17, 18, 26, 27 y 30). Esto se origina porque para la miniespecificación el primer elemento del arreglo tiene el subíndice 1, pero para el programa el primer elemento tiene el subíndice 0; para la miniespecificación el segundo elemento tiene el subíndice 2, y para el programa es 1. Si se observa, el subíndice del programa es igual al subíndice de la miniespecificación, menos 1. Al colocar en el programa el subíndice como i-1 homologamos el funcionamiento de la miniespecificación y del programa, y de esa forma el resto de la miniespecificación sigue funcionando sin cambio, además que el usuario no se percata del cambio.

3. Compile el programa utilizando la siguiente línea.

```
vbc C5Promedios.vb
```

4. Si no genera errores, ejecute el programa de la siguiente forma.

```
C5Promedios.exe
```

Si todo ocurrió como se esperaba, la salida será la siguiente:

Salida

```
Valor para el elemento 1: 1
Valor para el elemento 2: 2
Valor para el elemento 3: 3
Valor para el elemento 4: 4
Valor para el elemento 5: 5
Número 1 es menor al promedio
Número 2 es menor al promedio
Número 3 es mayor o igual al promedio
Número 4 es mayor o igual al promedio
Número 5 es mayor o igual al promedio
Suma: 15
Promedio: 3
```

FIN DEL EJERCICIO ✱

Ejercicio 10.06

Desarrollando un programa de regla de tres simple usando Visual Basic.NET

Utilizando Visual Basic.NET, realice las siguientes actividades:

- ❒ Desarrolle el código fuente.
- ❒ Asigne como nombre del programa TresSimple.vb
- ❒ Compílelo.
- ❒ Ejecútelo.

Elabore un programa que realice el cálculo de una regla de tres simple. Como sabe, dicha regla permite calcular la proporción de un número respecto a otro que actúa como referencia de la proporción. Este programa contesta preguntas del tipo ¿qué porcentaje de 200 es 135? El 200 sería el número base de la proporción, mientras que el 135 sería la proporción. La solución se calcula multiplicando la proporción por 100, y dividiéndolo entre el número base de la proporción.

El programa deberá mostrar el resultado del cálculo.

FIN DEL EJERCICIO ✱

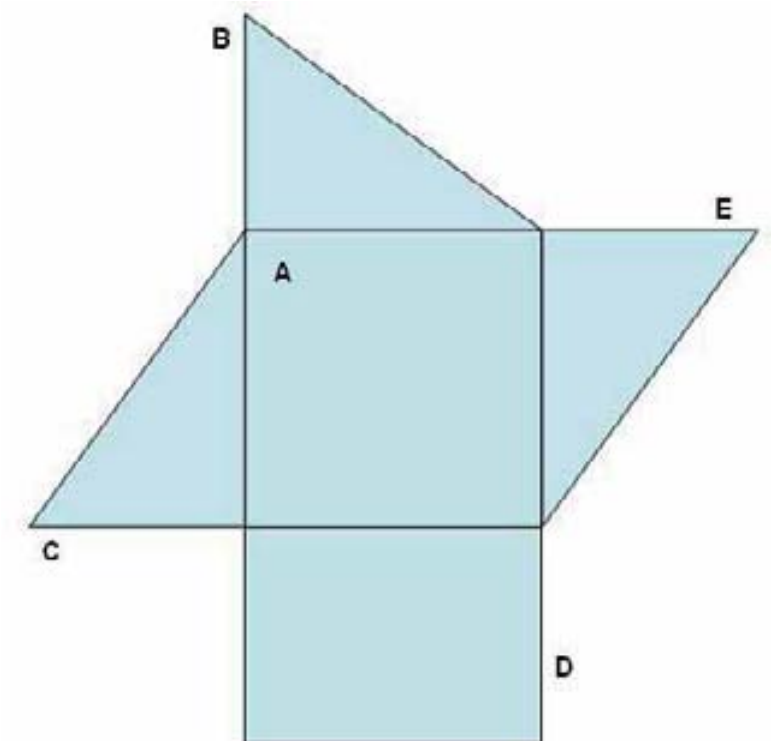
Ejercicio 10.07

Desarrollando un programa para el cálculo del área de un polígono irregular usando Visual Basic.NET

Utilizando Visual Basic.NET, realice las siguientes actividades:

- ❒ Desarrolle el código fuente.
- ❒ Asigne como nombre del programa Poligono.vb
- ❒ Compílelo.
- ❒ Ejecútelo.

Observe la siguiente figura.



El programa debe calcular el área del polígono. La figura se compone de un cuadrado perfecto (A), tres triángulos rectángulos iguales (B, C, E), cuya base mide lo mismo que uno de los lados del cuadrado, y un rectángulo cuyo ancho mide lo mismo que uno de los lados del cuadrado, mientras que la altura mide lo mismo que la altura de los triángulos. Se pide que haga un programa que calcule el área total.

Por cultura general, sabemos que el área de un rectángulo se calcula multiplicando su base por su altura; el área de un triángulo rectángulo se calcula multiplicando su base por su altura, y luego dividiendo el resultado entre dos; finalmente, el área de un cuadrado se calcula elevando al cuadrado la medida de uno de sus lados.

FIN DEL EJERCICIO ✱

Ejercicio 10.08

Desarrollando un programa con bucles, condicionales y arreglos usando Visual Basic.NET

Utilizando Visual Basic.NET, realice las siguientes actividades:

- ❒ Desarrolle el código fuente.
- ❒ Asigne como nombre del programa Descuento.vb
- ❒ Compílelo.
- ❒ Ejecútelo.

Una tienda departamental ofrece descuentos directos sobre las compras que los clientes realizan.

El descuento normal va en función al tipo de cliente que la persona sea: A los clientes normales se les ofrece un 10 de descuento, y a los clientes premier un 15 de descuento.

Actualmente se tiene una promoción: en la compra de 5 artículos, se ofrecerá un descuento adicional al normal. Si el monto de la compra no excede de 1,000 pesos, se agrega un 5 de descuento; si el monto de la compra está entre 1,000 y 2,000 pesos, se agrega un descuento del 10 ; si la compra es mayor a 2,000 pesos, entonces el descuento adicional será del 15 .

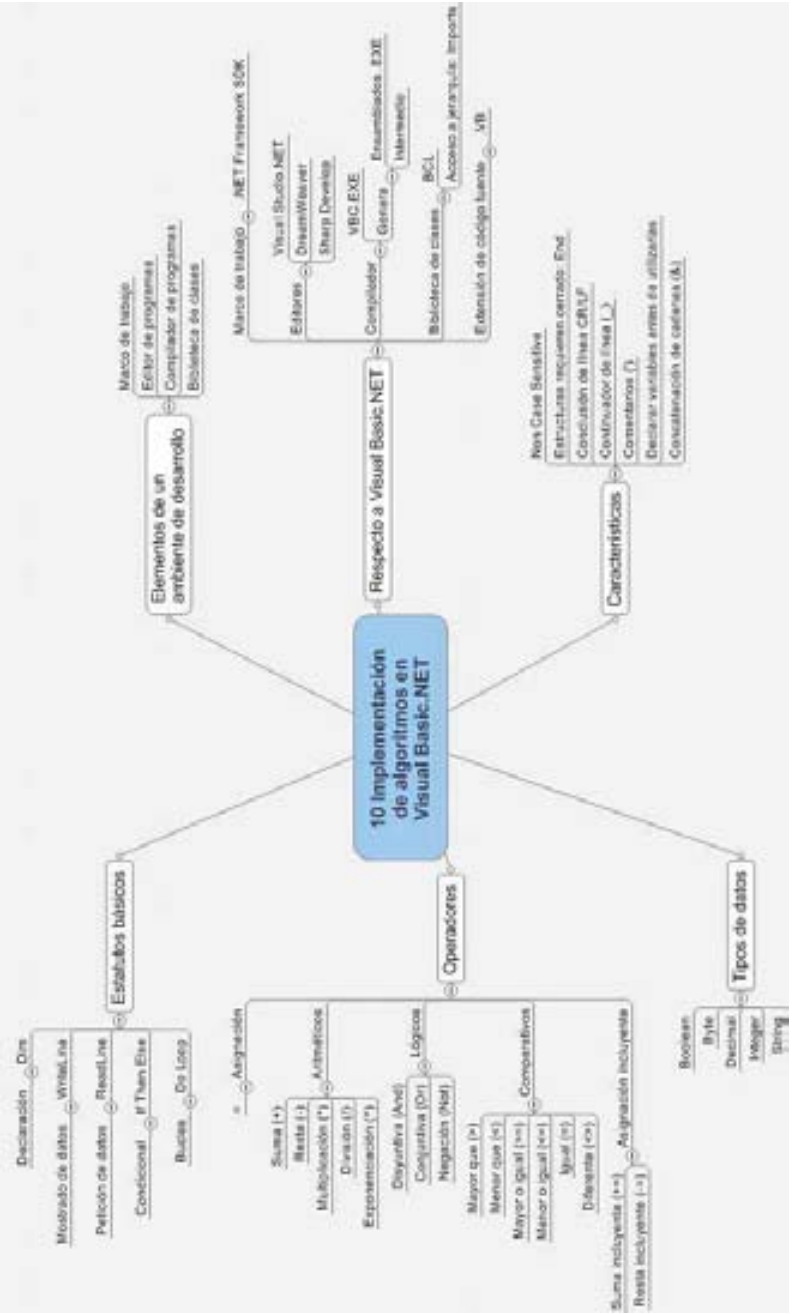
En ese sentido, el menor descuento que se puede obtener es cuando un cliente normal compra menos de 1,000 pesos (15), y el máximo descuento que se puede obtener es cuando un cliente premier compra más de 2,000 pesos (30).

El programa deberá pedir la información de los precios de 5 productos, y determinar el monto de descuento aplicable.

Utilice arreglos.

FIN DEL EJERCICIO ✱

Mapa mental del capítulo



Terminología

Ambiente de desarrollo, 269
Assembly, 282
Base Class Library, 271
BCL, 271
Biblioteca de clases base, 271
Biblioteca de clases, 270
Código intermedio de .NET, 282
Compilador de programas, 270
Editor de programas, 269
Ensamblado, 282
Ensamblados, 271
Espacio de nombres, 272
Just in Time Compiler, 271
Macromedia DreamWeaver, 271
Marco de trabajo, 269
Microsoft .NET Framework Redistributable Package, 270
Microsoft .NET Framework SDK, 270
Microsoft .NET Framework, 270
Microsoft Intermediate Language, 282
Microsoft Visual Studio.NET, 271
MSIL, 282
Non case-sensitive, 273
Sharp Develop, 271
vbc, 271

Preguntas

10.1 Tome tiempo de los siguientes aspectos:

10.1.1 ¿Tiempo en que se desarrollaron los casos 6, 7 y 8 en Visual Basic.NET?

10.1.2 ¿Veces en que fue necesario corregir cada uno de los programas 6, 7 y 8, a fin de corregir el programa por errores de sintaxis?

10.1.3 ¿Qué tan fácil fue desarrollar los casos 6, 7 y 8 en Visual Basic.NET, en su opinión?

10.1.4 ¿Qué tan útiles fueron las miniespecificaciones que desarrolló, para efectos de construir su código en Visual Basic.NET? ¿Requirieron ajustes?

Estas preguntas se repetirán para cada lenguaje que vaya explorando. Las respuestas a estas preguntas serán útiles para un comparativo de lenguajes al finalizar la lectura del libro.

Examen rápido

Parte 1 : Marque co n una cruz el inciso que considere que responde a la pregunta.

1) Es el marco de trabajo necesario para trabajar con programas desarrollados en Visual Basic.NET
(a) .NET Frame or
(b) J2SE
(c) No se requiere un marco de trabajo

2) El compilador de Visual Basic.NET es
(a) Bcc32
(b) Vbc.exe
(c) Javac

3) La biblioteca de clases que utiliza Visual Basic.NET se llama
(a) JFC
(b) BCL
(c) No tiene nombre

4) Al compilar un programa en Visual Basic.NET se genera un archivo de extensión
(a) E E
(b) Class
(c) OBJ

5) Al compilar un programa en Visual Basic.NET el archivo que se genera es llamado
(a) assembly
(b) bytecode
(c) machine code

Parte 2: Coloque en la línea la letra «F» si la sentencia es falsa, y «V» si la sentencia es verdadera.

6) El compilador de Visual Basic.NET es en dos fases

_____ **7)** Visual Basic.NET soporta bucles de tipo Until.

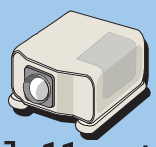
_____ **8)** NetBeans es una herramienta de desarrollo asociada a Visual Basic.NET

_____ **9)** La biblioteca de clase de Visual Basic.NET se compone de archivos de extensión .H

_____ **10)** En Visual Basic.NET para tener acceso a la biblioteca de clases es necesario referir el nombre físico de la biblioteca, dentro del programa, usando Imports

11

Implementación de algoritmos en C#



lp11.ppt

Objetivos: Aprenderá a implementar algoritmos utilizando el lenguaje de programación C# .

1. Aprenderá cuáles son los requerimientos de ambiente para desarrollar en C#.
2. Aprenderá las generalidades del lenguaje C#, así como la estructura básica de sus programas.
3. Aprenderá a realizar cambios de estado, peticiones de datos, muestra de datos, condicionales y bucles usando C#.
4. Aprenderá a compilar y ejecutar programas desarrollados en C#.

www.detodoprogramacion.com

Contenido

Implementación de algoritmos en C# 302
Ambiente de desarrollo 302
Estructura básica de un programa 305
Características generales del lenguaje 305
Tipos de datos 306
Declaración de variables y arreglos 306
Operadores principales 307
Cambios de estado 308
Mostrado de datos 309
Peticiones de datos 309
Estructuras de decisión 311
Estructuras de control 312
Compilación y ejecución de programas 315
Codificación de C1Mensaje en C# 315
Codificación de C2Niveles en C# 317
Codificación de C3Tabla en C# 319
Codificación de C4MultiTabla en C# 321
Codificación de C5Promedios en C# 324
Desarrollando un programa de regla de tres simple usando C# 327
Desarrollando un programa para el cálculo del área de un polígono irregular usando C# 328
Desarrollando un programa con bucles, condicionales y arreglos usando C# 329
Mapa mental del capítulo 330
Terminología 331
Preguntas 331
Examen rápido 333

Implementación de algoritmos en C#

C es un lenguaje de propósito general orientado a objetos, que permite desarrollar aplicaciones bajo la plataforma de desarrollo de .NET Frame or .

Por estar basado en .NET Frame or , le aplican las mismas condiciones de ambiente de desarrollo que a Visual Basic.NET.

Ambiente de desarrollo

El ambiente de desarrollo del lenguaje C se compone como sigue.

- *arco de trabajo.*

  Para el desarrollo de C es necesario el marco de trabajo denominado Microsoft .NET Frame or , disponible desde la página eb de Microsoft

Corporation. Para labores de desarrollo, es necesario instalar Microsoft.NET Frame or Redistributable Pac age, y luego Microsoft.NET Frame or SDK, que puede descargarse de forma gratuita en www.microsoft.com. Deberá instalarlos para poder probar los programas en C .

Las pruebas de este libro se hicieron con la versión 2.0 de .NET Frame or SDK. Si es su caso, la ruta de herramientas del marco de trabajo puede agregarse desde línea de comandos, ejecutando lo siguiente:

```
PATH="%PATH%";"C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727"
```

Puede variar dependiendo la versión instalada de .NET Frame or (v1.0.3705, v1.1.4322, v2.0.50215) o del sistema operativo (WINNT).

- *Editor de programas.*

  Aunque se pueden editar programas C en cualquier editor de texto plano, la herramienta más sofisticada para la programación en dicho lenguaje es Microsoft Visual Studio.NET. Otras opciones que se tienen es Macromedia Dream eaver, Sharp Develop, entre otros.

- *ompilador de programas.*

  El compilador de programas en C es el programa csc.exe, que viene incluido en Microsoft.NET Frame or SDK. Este compilador produce archivos de extensión EXE, que a pesar de la extensión, no son ejecutables puros. Los archivos EXE generados por el compilador son código intermedio, a los que se les da el nombre de *ensamblados*. Al ejecutarse por primera vez los programas EXE desarrollados en C , un elemento del .NET Frame or denominado *ust n ime ompiler* realizará la fase de síntesis del proceso de compilación, creando finalmente el código máquina.

- *iblioteca de clases.*

  La biblioteca de clases de Microsoft .NET Frame or está formada por más de 6,000 clases que permiten prácticamente cualquier cosa. Esta biblioteca recibe el nombre de *iblioteca de clases base* (*ase lass ibrary*), que se compone por un conjunto de archivos DLL utilizables por todos los lenguajes .NET.

  En el caso de C la biblioteca de clases se instala y está disponible en el equipo al instalar .NET Frame or , por lo que no es necesario hacer referencia a los archivos físicos DLL que contienen las clases.

  Las clases contenidas en los DLL se organizan a través de una jerarquía de clases. Para tener acceso a la funcionalidad de una clase desde un programa es necesario especificar toda su jerarquía al momento de invocarla. Por ejemplo, si deseamos invocar las clases para el manejo de entrada y salida por consola, es necesario invocarlas de la siguiente manera: System.Console.Read() y System.Console.Write(). System es la jerarquía,

www.detodoprogramacion.com

Console es la clase, y Read() y Write() son los métodos que nos dan realmente la funcionalidad de escritura y lectura.

Como es un poco molesto tener que estar repitiendo como prefijo la jerarquía de clase (System) podemos utilizar un espacio de nombre al inicio el programa, para obviar la referencia. Un *espacio de nombres* es un medio para la referencia lógica de la funcionalidad contenida en una biblioteca de clases, permitiendo el acceso simplificado a las clases. En C podemos establecer espacios de nombre escribiendo al inicio del programa líneas similares a esta.

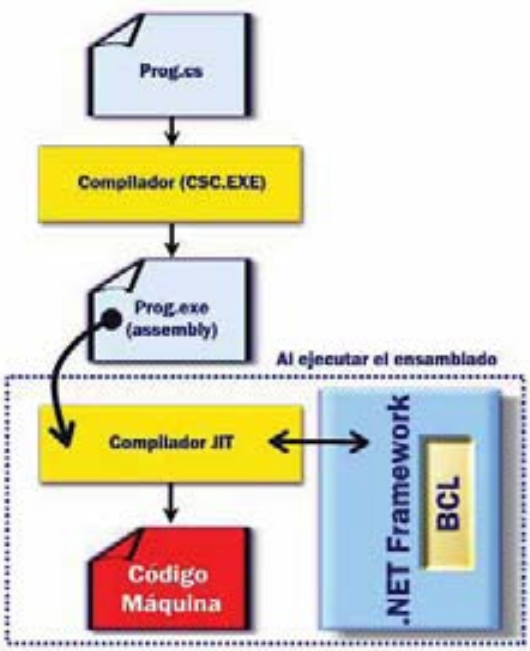
```
using System;
```

Con lo cual las instrucciones System.Console.Read() y System.Console.Write() podrían referirse simplemente como Console.Write() y Console.Read().

El uso de espacios de nombres nos permite obviar la especificación de la jerarquía de clases al momento de invocar la funcionalidad de las bibliotecas.

Figura 11.01

Proceso de compilación y ejecución en C#



Estructura básica de un programa

La estructura base de todo programa en el lenguaje C es la siguiente.

```
1   using System;
2
3   class  ombrePrograma
4   {
5           static void Main()
6           {
7
8                   ...
9
10          }
11  }
```

Por convencionalismo, se debe colocar como nombre de la clase el nombre del programa. Nuestro procedimiento principal deberá incluirse en el espacio que ocupa la línea 8 de la ilustración.

Características generales del lenguaje

A C le aplican las siguientes reglas generales básicas.

1. Es sensible a mayúsculas y minúsculas (es *case sensitive*), por lo que el uso de mayúsculas y minúsculas son relevantes para el compilador.
2. Los bloques de código se encierran entre llaves “ ”.
    a. Las llaves van siempre por pares.
    b. Todas las llaves que se abran deben cerrarse.
    c. La última llave en abrirse, será siempre la primera en cerrarse.
3. Se considera un bloque de código la presencia de dos o más líneas de código.
4. Las líneas en C se terminan con punto y coma “;”.
. Las palabras reservadas, intrínsecas del lenguaje C , se escriben en minúsculas.
6. Los comentarios en C se especifican colocando doble diagonal “ ” al inicio de una línea.
. Para utilizar variables, antes deben declararse.

Tipos de datos

Los tipos de datos básicos del lenguaje C son los siguientes.

Tipo	Consumo en Bytes	Dominio de tipo
bool		r e o alse
byte	1 byte	0 a 255 (sin signo) .
decimal	16 bytes	0 a 79.228.162.514.264.337.593.543.950.335 sin separador decimal; 0 a 7,9228162514264337593543950335 con 28 posiciones a la derecha del signo decimal; el número más pequeño distinto de cero es 0,0000000000000000000000000001 (1E 28).
int	4 bytes	2.147.483.648 a 2.147.483.647
String	Clase (2 bytes por caracter, aprox)	Cadenas de caracteres.

Declaración de variables y arreglos

La forma de declarar variables en el lenguaje C es utilizando la siguiente sintaxis.

tipo ariable [= *E presión*];

Donde *ariable* es un nombre de variable válido, y *tipo* es un tipo de datos reconocido por el lenguaje. En C se puede asignar un valor de inicio al momento de declarar una variable, asignando una *E presión*, con posterioridad al nombre de la variable.

A continuación unos ejemplos.

```
int i;
int i=1;
String Mensaje;
```

Equivalencias

Miniespecificación	Diagrama de flujo	C#
(i,i[1-n])	i = 1	int i=1;
(Suma,d[1-n])	Suma = 0	decimal Suma=0;

Los arreglos se definen en el lenguaje C utilizando new. La sintaxis es la siguiente.

tipo[] *ariable ne tipo Elementos* ;

Donde *ariable* es un nombre de variable válido, *tipo* es un tipo de datos reconocido por el lenguaje, y *Elementos* es el número de elementos del arreglo. En C los arreglos son de base cero, es decir, que los subíndices son 0, 1, 2, 3, y así. Declarar un arreglo como Numeros(3), genera un arreglo de 3 elementos con los subíndices 0,1,2 .

A continuación unos ejemplos.

```
int[ ] Ventas = new int[3];
int[ ] Ventas = new int[ ] {20000,15000};
```

Operadores principales

Los operadores principales del lenguaje C , aplicables a los ejercicios tratados en el libro, son los siguientes.

Tipo	Operador	Nombre
Asignación	=	Igual
Aritmético	+	Suma/Concatenación de cadenas
	-	Resta
	*	Multiplicación
	/	División

	^	Exponenciación
Lógico		Disyuntiva
	||	Conjuntiva
	!	Negación
Comparativos	>	Mayor que
	<	Menor que
	>=	Mayor o igual que
	<=	Menor o igual que
	==	Igual a
	<>	Distinto a
Asignación incluyente	+=	Suma incluyente
	-=	Resta incluyente

Cambios de estado

Los cambios de estado en el lenguaje C se realizan utilizando el operador de asignación (=).

ariable = E presión

Donde *ariable* es el nombre de una variable previamente declarada y *E presión* es el valor que se le asigna a la variable. Estos son unos ejemplos.

```
AreaCirculo = Lado^2;
i = i + 1;
i += 1;
```

En el ejemplo, las últimas dos líneas hacen lo mismo.

Equivalencias

Miniespecificación	Diagrama de flujo	C#
Promedio ← Suma/5		Promedio = Suma/5;

Suma ← Suma + Entrada(i)	Suma = Suma + Entrada(i)	Suma += Entrada(i);
i ← i + 1	i = i + 1	i += 1;

Mostrado de datos

El mostrado de datos se realiza en el lenguaje de C a través de las órdenes Console.Write y Console.WriteLine. La terminación "Line" de los comandos aplica si deseamos que al final de la instrucción se incluya un salto de línea.

```
Console.Write("E presión ");
```

Donde *E presión* es la expresión a mostrar.

Ejemplo.

```
Console.Write("Dame un número: ");
```

Equivalencias

Miniespecificación
« "Suma: " + Suma
Diagrama de flujo
PUT "Suma: " + Suma
C#
Console.WriteLine("Suma:"+Suma);

Peticiones de datos

Las peticiones de datos se realizan en el lenguaje de C a través de las órdenes Console.Read y Console.ReadLine.

Console.Read sirve para leer los datos. La terminación "Line" de los comandos aplica si deseamos que al final de la instrucción se incluya un salto de línea.

```
ariable = Console.ReadLine();
```

Donde *ariable* es la variable en la que se almacenará lo que el usuario introduzca a la interfaz.

Es muy importante hacer notar que ReadLine adquiere datos de tipo String. En caso de que se estén preguntando números, es necesario realizar la conversión al asignar valores a una variable numérica.

Si se pregunta un dato tipo...	...Se debe preguntar de la siguiente forma
byte	byte.Parse(*tring*)
decimal	decimal.Parse(*tring*)
int	int.Parse(*tring*)

Esto aplica para todos los datos numéricos.

Ejemplos.

```
Mensaje=Console.ReadLine();
Numero = int.Parse(Console.ReadLine());
```

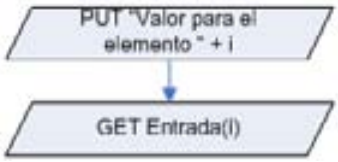
Generalmente las peticiones de datos se realizan a solicitud, por lo que es común asociar una petición con un mostrado de datos previo, quedando como sigue.

```
Console.Write("E presión");
ariable = Console.ReadLine();
```

Equivalencias

Miniespecificación
« "Valor para el elemento " + i
» Entrada(i)

Diagrama de flujo



C#

```
Console.WriteLine("Valor para el elemento " + i);
Entrada(i) = int.Parse(Console.ReadLine());
```

Estructuras de decisión

Las estructuras de decisión se realizan en el lenguaje C a través de la estructura if.

```
If ondición
{
        nstrucciones por verdadero
[   } else {
        nstrucciones por falso ]
}
```

Donde *ondición* es la expresión lógica a evaluar. En caso de que *ondición* sea True, se ejecutará el bloque de código que contiene las *nstrucciones por verdadero*. De manera opcional se puede especificar un bloque de código a ejecutar si *ondición* resuelve por falso, utilizando else. En ese caso, se ejecutarían las *nstrucciones por falso*. La estructura if siempre debe terminar.

Ejemplos.

```
if Monto > LimiteCredito
{
        Venta = "Rechazada";
}
else
```

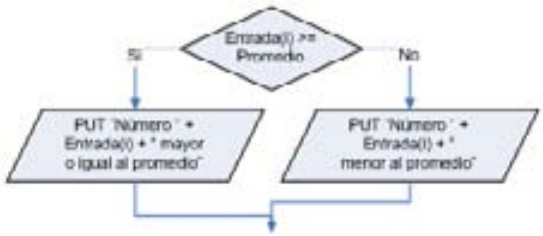
```
{
        Venta = “Aprobada”;
}
```

Equivalencias

Miniespecificación

```
Si Entrada(i) >= Promedio Entonces
        « “Numero ” + Entrada(i) + “ mayor o igual al promedio”
De lo contrario
        « “Numero ” + Entrada(i) + “ menor al promedio”
Fin Si
```

Diagrama de flujo



C#

```
if  Entrada(i) >= Promedio
{
  Console.WriteLine(“Numero ” + Entrada(i) + “ mayor o igual al promedio”);
}
else
{
    Console.WriteLine(“Numero ” + Entrada(i) + “ menor al promedio”);
}
```

Estructuras de control

Las estructuras de control se realizan en el lenguaje C a través de la estructura do while.

Es importante señalar la forma en que se manejan las estructuras de control con condición al final en C , dado que las miniespecificaciones que pretendemos codificar están diseñadas de esa forma.

Para empezar, una estructura de control con condición al final representa una estructura de control de tipo until (hasta que). Dado que C no posee estructura until (hasta que), se deberá especificar una condición que mantenga el proceso ejecutándose, mientras la condición se esté cumpliendo.

Supongamos que tenemos un bucle del 1 al 10. En las estructuras until la condición provocaba la terminación del ciclo, de ahí que el proceso se mantenía ejecutando hasta que 1 > 10. Con while la condición tendría que cambiarse a 1 <=10, es decir, mientras sea menor o igual a 10.

```
do
{
        nstrucciones

while (  ondición);
```

Donde *nstrucciones* es el bloque de código que se estará repitiendo hasta que *ondición* se cumpla. Todo lo que esté entre do y while se considera código controlado. Es importante no olvidar colocar y cambiar el estado de la variable de trabajo, dentro del código de control.

Ejemplos.

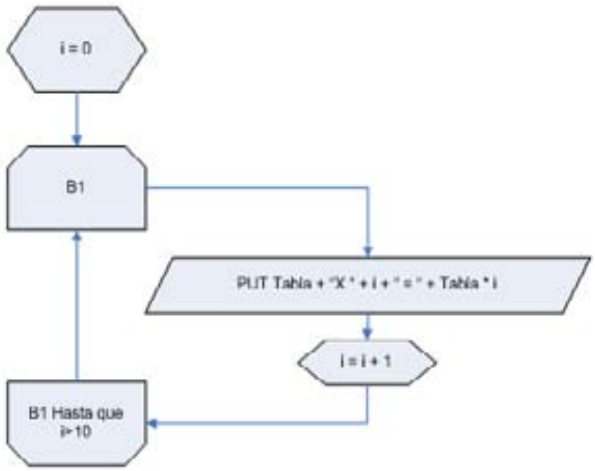
```
i = 1;
do
{
        Console.WriteLine("Valor del contador: " + i);
        i += 1;
}
while (i <= 5);
```

Equivalencias

Miniespecificación
i ← 1
Inicio

```
        « Tabla + "X" + i + "=" + Tabla*i
        i ← i + 1
Hasta que i > 10
```

Diagrama de flujo



C#

```
i = 1;
do
{
        Console.WriteLine(Tabla + "X" + i + "=" + Tabla*i);
        i += 1;
}
while (i <= 10);
```

Vea cómo la condición del bucle cambia a la condición opuesta, dado que cambiamos de un enfoque until a un enfoque while.

Aunque no manejamos estructuras de control con condición al inicio en este libro, proporcionamos la sintaxis de su implementación en C .

```
while (  ondición)
{
        nstrucciones
```

Donde *nstrucciones* es el bloque de código que se estará repitiendo mientras que *ondición* se cumpla por verdadero.

Compilación y ejecución de programas

La compilación de los programas se realiza utilizando el compilador de línea de comandos de C , llamado csc.exe. La sintaxis básica de compilación es la siguiente.

```
csc Programa.cs
```

Donde *Programa* es el programa a compilar.

El comando se ejecuta desde la línea de comandos. Es importante no omitir la extensión de los programas desarrollados en C (.cs), dado que la compilación generaría error.

Al concluir la compilación sin errores, se habrá creado un programa de extensión .exe, que se trata de *código intermedio de . E* , llamado también (*icrosoft ntermediate anguage*), más conocido aún como *ensamblado* (*assembly*); sólo se podrá ejecutar el ensamblado en equipos con .NET Frame or instalado.

El programa podrá ejecutarse desde la línea de comandos, invocando el programa ejecutable que se generó.

```
Programa
```

Donde *Programa* es el nombre del archivo .exe que se desea ejecutar.

Al ejecutar el código ensamblado, el Just In Time Compiler de .NET Frame or detecta que no se trata de código máquina, realiza la fase de síntesis del proceso de compilación a partir del código intermedio de .NET, generando el código máquina optimizado para el equipo desde el cual se intentó la ejecución, mostrando finalmente los resultados del programa.

Ejercicio 11.01

Codificación de C1Mensaje en C#

En el presente ejercicio se codificará el primer caso de estudio, utilizando el lenguaje C .

1. Utilice un editor de texto plano (*loc de notas*, por ejemplo), para escribir el código fuente llamado C1Mensaje.cs; el archivo deberá guardarse en el directorio de trabajo del libro (C:\APIP).

Este programa tiene como finalidad comprobar técnicas y herramientas de desarrollo. Lo que el programa debe hacer es preguntar por un mensaje de saludo, para posteriormente mostrar el mensaje que se le proporcionó.

Miniespecificación

```
1   INICIO:
2           (Mensaje, x(30))
3           « "Escribe un mensaje a mostrar: "
4           » Mensaje
5           « Mensaje
6   :FIN
```

Codificación de **C1Mensaje.cs**

```
1   using System;
2
3   class C1Mensaje
4   {
5           static void Main()
6           {
7
8                   String Mensaje;
9                   Console.Write("Escribe un mensaje a mostrar: ");
10                  Mensaje= Console.ReadLine();
11                  Console.WriteLine(Mensaje);
12
13          }
14  }
```

2. Compile el programa utilizando la siguiente línea.

```
csc C1Mensaje.cs
```

3. Si no genera errores, ejecute el programa de la siguiente forma.

```
C1Mensaje.exe
```

Si todo ocurrió como se esperaba, la salida será la siguiente:

Entrada	Escribe un mensaje a mostrar: Aprenda
Salida	Aprenda

FIN DEL EJERCICIO ✱

Ejercicio 11.02

Codificación de C2Niveles en C#

En el presente ejercicio se codificará el segundo caso de estudio, utilizando el lenguaje C .

1. Utilice un editor de texto plano (*loc de notas*, por ejemplo), para escribir el código fuente llamado C2Niveles.cs; el archivo deberá guardarse en el directorio de trabajo del libro (C:\APIP).

Se desea saber la disponibilidad del servicio de Internet para una persona, expresado en horas de disponibilidad.

Dicho nivel de servicio depende del grado de disponibilidad que ofrece la computadora que utiliza, así como del grado de disponibilidad del enlace a Internet que utiliza la máquina.

El horario de labores de la persona que utiliza la computadora y el enlace a Internet es de 8 horas.

Para poder realizar los cálculos adecuadamente, tome en cuenta que la disponibilidad de un sistema de dispositivos (dos o más dispositivos que trabajan de forma conjunta) se calcula multiplicando los porcentajes de grado de disponibilidad de cada uno de ellos, y el resultado siempre será menor a la disponibilidad del menor.

Para calcular el tiempo de disponibilidad de un servicio (en este caso el acceso a la información publicada en Internet), sólo basta con multiplicar el tiempo en que es requerido el servicio por la disponibilidad del sistema de dispositivos.

Miniespecificación

```
1   INICIO:
2           (DC,d[0-1])
3           (DE,d[0-1])
4           (TL,d[0-1])
5           (DS,d[0-1])
```

```
6           (DH,d[0-1])
7           « "Grado de disponibilidad de la computadora:"
8           » DC
9           « "Grado de disponibilidad del enlace:"
10          » DE
11          « "Tiempo laborable en horas:"
12          » TL
13          DS ← DC * DE
14          DH ← DS * TL
15          « "La disponibilidad del servicio de Internet es de " + DH + " horas"
16  :FIN
```

Codificación de **C2Niveles.cs**

```
1   using System;
2
3   class C2Niveles
4   {
5           static void Main()
6           {
7
8                   decimal DC;
9                   decimal DE;
10                  decimal TL;
11                  decimal DS;
12                  decimal DH;
13
14                  Console.Write("Grado de disponibilidad de la computadora: ");
15                  DC = decimal.Parse(Console.ReadLine());
16                  Console.Write("Grado de disponibilidad del enlace: ");
17                  DE = decimal.Parse(Console.ReadLine());
18                  Console.Write("Tiempo laborable en horas: ");
19                  TL = decimal.Parse(Console.ReadLine());
20
21                  DS= DC*DE;
22                  DH= DS*TL;
23
24                  Console.WriteLine("La disponibilidad del servicio de Internet es de "+
25                          DH +" horas");
26
27          }
28  }
```

2. Es importante analizar que al preguntar los datos compile el programa utilizando la siguiente línea.

```
csc C2Niveles.cs
```

3. Si no genera errores, ejecute el programa de la siguiente forma.

```
C2Niveles.exe
```

Si todo ocurrió como se esperaba, la salida será la siguiente:

Entrada	Grado de disponibilidad de la computadora: .98
Entrada	Grado de disponibilidad del enlace: .85
Entrada	Tiempo laborable en horas: 8
Salida	La disponibilidad del servicio de internet es de 6.6640 horas

FIN DEL EJERCICIO ✱

Ejercicio 11.03

Codificación de C3Tabla en C#

En el presente ejercicio se codificará el tercer caso de estudio, utilizando el lenguaje C .

1. Utilice un editor de texto plano (*loc de notas*, por ejemplo), para escribir el código fuente llamado C3Tabla.cs; el archivo deberá guardarse en el directorio de trabajo del libro (C:\APIP).

El programa debe mostrar cualquier tabla de multiplicar que le solicitemos. La única restricción es que la tabla deberá ser de un número mayor o igual a 1, y menor o igual a 10.

Si se pide una tabla de un número que no esté entre el 1 y el 10, el programa debe indicar, mediante un mensaje, que no puede continuar.

Miniespecificación

```
1   INICIO:
2           (i,i[0-n])
3           (Tabla,i[1-10])
4           « "Qué tabla deseas: "
5           » Tabla
6           Si Tabla >= 1 ∧ Tabla <=10 Entonces
7                   i ← 1
```

```
8                       Inicio
9                               « Tabla + “ X” + i + “=” + Tabla * i
10                              i ← i + 1
11                      Hasta que i > 10
12              De lo contrario
13                      « “Válido sólo del 1 al 10”
14              Fin Si
15      :FIN
```

Codificación de **C3Tabla.cs**

```
1   using System;
2
3   class C3Tabla
4   {
5           static void Main()
6           {
7
8                   int Tabla;
9                   int i=1;
10
11                  Console.Write("Qué tabla deseas: ");
12                  Tabla =  int.Parse(Console.ReadLine());
13
14                  if(Tabla >= 1 & Tabla <=10)
15                  {
16                          i=1;
17                          do
18                           {
19                                  Console.WriteLine(Tabla + " X " + i + " = "+ Tabla * i );
20                                  i+=1;
21                           }
22                          while (i <=10);
23                  }
24                  else
25                  {
26                          Console.Write("Válido solo del 1 al 10");
27                  }
28
29          }
30  }
```

2. Compile el programa utilizando la siguiente línea.

```
csc C3Tabla.cs
```

3. Si no genera errores, ejecute el programa de la siguiente forma.

```
C3Tabla.exe
```

Si todo ocurrió como se esperaba, la salida será la siguiente:

Entrada	Que tabla deseas: 5
Salida	5 X 1 = 5
	5 X 2 = 10
	5 X 3 = 15
	5 X 4 = 20
	5 X 5 = 25
	5 X 6 = 30
	5 X 7 = 35
	5 X 8 = 40
	5 X 9 = 45
	5 X 10 = 50

Entrada	Que tabla deseas: 25
Salida	Válido sólo del 1 al 10

FIN DEL EJERCICIO ✱

Ejercicio 11.04

Codificación de C4MultiTabla en C#

En el presente ejercicio se codificará el cuarto caso de estudio, utilizando el lenguaje C .

1. Utilice un editor de texto plano (*loc de notas*, por ejemplo), para escribir el código fuente llamado C4MultiTabla.cs; el archivo deberá guardarse en el directorio de trabajo del libro (C:\APIP).

El programa debe mostrar todas las tablas de multiplicar del 1 al 5, incluyendo un encabezado por tabla.

Miniespecificación

```
1   INICIO:
2           (i,i[0-n])
3           (y,i[0-n])
4           i ← 1
5           Inicio
6                   « "TABLA: " + i
7                   y ← 1
8                   Inicio
9                           « i + " X" + y + "=" + i * y
10                          y ← y + 1
11                  Hasta que y > 10
12                  i ← i + 1
13          Hasta que i > 5
14  :FIN
```

Codificación de **C4MultiTabla.cs**

```
1   using System;
2
3   class C4MultiTabla
4   {
5           static void Main()
6           {
7
8                   int i;
9                   int y;
10
11                  i = 1;
12                  do
13                  {
14                          Console.WriteLine("Tabla: "+i);
15                          y = 1;
16                          do
17                          {
18                                  Console.WriteLine(i + " X " + y + " = "+ i*y );
19                              y+=1;
20                          }
21                          while (y <=10 );
22                          i+=1;
23                  }
24                  while (i <= 5);
25
26          }
27  }
```

2. Compile el programa utilizando la siguiente línea.

```
csc C4MultiTabla.cs
```

3. Si no genera errores, ejecute el programa de la siguiente forma.

```
C4MultiTabla.exe
```

Si todo ocurrió como se esperaba, la salida será la siguiente:

Salida

```
Tabla: 1
1 X 1 = 1
1 X 2 = 2
1 X 3 = 3
1 X 4 = 4
1 X 5 = 5
1 X 6 = 6
1 X 7 = 7
1 X 8 = 8
1 X 9 = 9
1 X 10 = 10
Tabla: 2
2 X 1 = 2
2 X 2 = 4
2 X 3 = 6
2 X 4 = 8
...

5 X 7 = 35
5 X 8 = 40
5 X 9 = 45
5 X 10 = 50
```

FIN DEL EJERCICIO ✱

Ejercicio 11.05

Codificación de C5Promedios en C#

En el presente ejercicio se codificará el quinto caso de estudio, utilizando el lenguaje C .

1. Utilice un editor de texto plano (*loc de notas*, por ejemplo), para escribir el código fuente llamado C5Promedios.cs; el archivo deberá guardarse en el directorio de trabajo del libro (C:\APIP).

El programa debe pedir 5 números, para después mostrarlos indicando si están o no por encima de su promedio. Al final deberá enviarse un resumen en donde se indique la suma total de los números, y el promedio calculado.

Miniespecificación

```
INICIO:
        (i,i[0-n])
        (Entrada(5),i[0-n])
        (Suma,d[0-n])
        (Promedio,d[0-n])
        Suma ← 0
        i ← 1
        Inicio
                « "Valor para el elemento " + i + ": "
                » Entrada(i)
                Suma ← Suma + Entrada(i)
                i ← i + 1
        Hasta que i > 5
        Promedio ← Suma / 5
        i ← 1
        Inicio
                Si Entrada(i) >= Promedio Entonces
                    « "Número " + Entrada(i) + " mayor o igual al promedio: "
                De lo contrario
                    « "Número " + Entrada(i) + " menor al promedio: "
                Fin Si
                i ← i + 1
        Hasta que i > 5
        « "Suma: " + Suma
        « "Promedio: " + Promedio
:FIN
```

Codificación de **C5Promedios.cs**

```
1   using System ;
2
3   class C5Promedios
4   {
5           static void Main ( string[ ] args )
6           {
7
8                   int i;
9                   int[] Entrada = new int[5];
10                  int Suma;
11                  decimal Promedio;
12
13                  Suma=0;
14
15                  i=1;
16                  do
17                  {
18                          Console.Write("Valor para el elmento " + i + ": ");
19                          Entrada[i-1] = int.Parse(Console.ReadLine());
20                          Suma+=Entrada[i-1];
21                          i+=1;
22                  }
23                  while (i <=5 );
24
25                  Promedio=Suma/5;
26
27                  i=1;
28                  do
29                  {
30                          if(Entrada[i-1] >= Promedio)
31                          {
32                                  Console.WriteLine("Número " + Entrada[i-1] +
33                                          " mayor o igual al promedio");
34                          }
35                          else
36                          {
37                                  Console.WriteLine("Número " + Entrada[i-1] +
38                                          " menor al promedio");
39                          }
40                          i+=1;
41                  }
42                  while (i <=5 );
43
44                  Console.WriteLine("Suma: " + Suma );
45                  Console.WriteLine("Promedio: " + Promedio );
46
47          }
48  }
```

2. Es importante mencionar que este programa tiene discrepancias con la miniespecificación, provocadas por el cambio de enfoque en las estructuras de control. El hecho de cambiar las estructuras de tipo until, por estructuras de tipo while, hace que las condiciones que eran i > 5 en la miniespecificación (líneas 13 y 23) pasen a ser i<=5 en el programa (líneas 23 y 42).

3. En las llamadas al arreglo, en la miniespecificación se coloca como subíndice i (líneas 11, 17, 18 y 20), mientras que en el programa se refiere al subíndice como i-1 (líneas 19, 20, 30, 32, 37). Esto se origina porque para la miniespecificación el primer elemento del arreglo tiene el subíndice 1, pero para el programa el primer elemento tiene el subíndice 0; para la miniespecificación el segundo elemento tiene el subíndice 2, y para el programa es 1. Si se observa, el subíndice del programa es igual al subíndice de la miniespecificación, menos 1. Al colocar en el programa el subíndice como i-1 homologamos el funcionamiento de la miniespecificación y del programa, y de esa forma el resto de la miniespecificación sigue funcionando sin cambio, además que el usuario no se percata del cambio.

4. Compile el programa utilizando la siguiente línea.

```
csc C5Promedios.cs
```

5. Si no genera errores, ejecute el programa de la siguiente forma.

```
C5Promedios.exe
```

Si todo ocurrió como se esperaba, la salida será la siguiente:

Salida

```
Valor para el elemento 1: 1
Valor para el elemento 2: 2
Valor para el elemento 3: 3
Valor para el elemento 4: 4
Valor para el elemento 5: 5
Número 1 es menor al promedio
Número 2 es menor al promedio
Número 3 es mayor o igual al promedio
```

```
Número 4 es mayor o igual al promedio
Número 5 es mayor o igual al promedio
Suma: 15
Promedio: 3
```

FIN DEL EJERCICIO ✱

Ejercicio 11.06

Desarrollando un programa de regla de tres simple usando C#

Utilizando C , realice las siguientes actividades:

- ❒ Desarrolle el código fuente.
- ❒ Asigne como nombre del programa TresSimple.cs .
- ❒ Compílelo.
- ❒ Ejecútelo.

Elabore un programa que realice el cálculo de una regla de tres simple. Como sabe, dicha regla permite calcular la proporción de un número respecto a otro que actúa como referencia de la proporción. Este programa contesta preguntas del tipo ¿qué porcentaje de 200 es 135? El 200 sería el número base de la proporción, mientras que el 135 sería la proporción. La solución se calcula multiplicando la proporción por 100, y dividiéndolo entre el número base de la proporción.

El programa deberá mostrar el resultado del cálculo.

FIN DEL EJERCICIO ✱

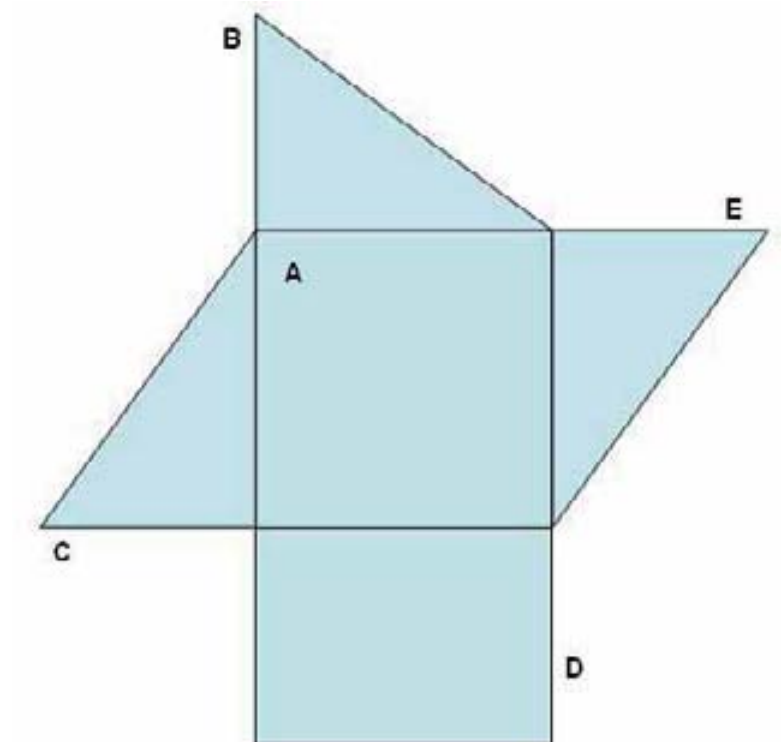
Ejercicio 11.07

Desarrollando un programa para el cálculo del área de un polígono irregular usando C#

Utilizando C , realice las siguientes actividades:

- ❒ Desarrolle el código fuente.
- ❒ Asigne como nombre del programa Poligono.cs .
- ❒ Compílelo.
- ❒ Ejecútelo.

Observe la siguiente figura.



El programa debe calcular el área del polígono. La figura se compone de un cuadrado perfecto (A), tres triángulos rectángulos iguales (B, C, E), cuya base mide lo mismo que uno de los lados del cuadrado, y un rectángulo cuyo ancho mide lo mismo que uno de los lados del

cuadrado, mientras que la altura mide lo mismo que la altura de los triángulos. Se pide que haga un programa que calcule el área total.

Por cultura general, sabemos que el área de un rectángulo se calcula multiplicando su base por su altura; el área de un triángulo rectángulo se calcula multiplicando su base por su altura, y luego dividiendo el resultado entre dos; finalmente, el área de un cuadrado se calcula elevando al cuadrado la medida de uno de sus lados.

FIN DEL EJERCICIO ✱

Ejercicio 11.08

Desarrollando un programa con bucles, condicionales y arreglos usando C#

Utilizando C , realice las siguientes actividades:

- ❒ Desarrolle el código fuente.
- ❒ Asigne como nombre del programa Descuento.cs .
- ❒ Compílelo.
- ❒ Ejecútelo.

Una tienda departamental ofrece descuentos directos sobre las compras que los clientes realizan.

El descuento normal va en función al tipo de cliente que la persona sea: A los clientes normales se les ofrece un 10 de descuento, y a los clientes premier un 15 de descuento.

Actualmente se tiene una promoción: en la compra de 5 artículos, se ofrecerá un descuento adicional al normal. Si el monto de la compra no excede de 1,000 pesos, se agrega un 5 de descuento; si el monto de la compra está entre 1,000 y 2,000 pesos, se agrega un descuento del 10 ; si la compra es mayor a 2,000 pesos, entonces el descuento adicional será del 15 .

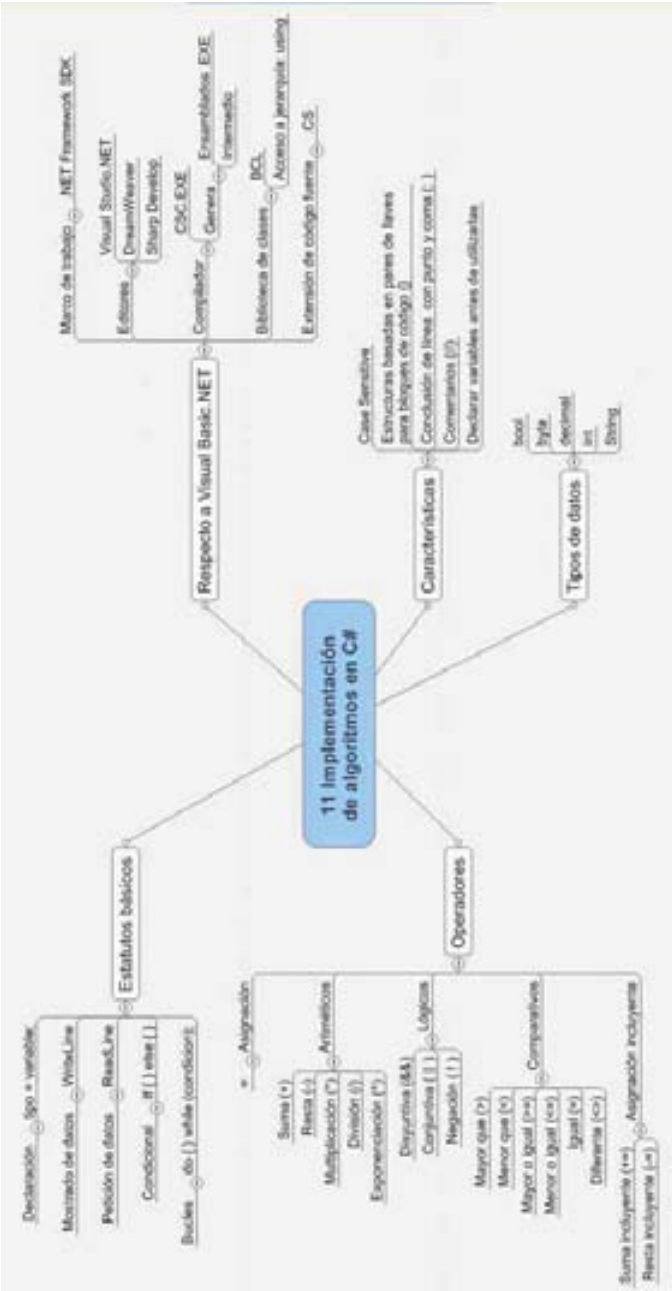
En ese sentido, el menor descuento que se puede obtener es cuando un cliente normal compra menos de 1,000 pesos (15), y el máximo descuento que se puede obtener es cuando un cliente premier compra más de 2,000 pesos (30).

El programa deberá pedir la información de los precios de 5 productos, y determinar el monto de descuento aplicable.

Utilice arreglos.

FIN DEL EJERCICIO ✱

Mapa mental del capítulo



Terminología

Case-sensitive, 305
csc.exe, 303
Parse, 310

Preguntas

11.1 Tome tiempo de los siguientes aspectos:

11.1.1 ¿Tiempo en que se desarrollaron los casos 6, 7 y 8 en C ?

11.1.2 ¿Veces en que fue necesario corregir cada uno de los programas 6, 7 y 8, a fin de corregir el programa por errores de sintaxis?

11.1.3 ¿Qué tan fácil fue desarrollar los casos 6, 7 y 8 en C , en su opinión?

11.1.4 ¿Qué tan útiles fueron las miniespecificaciones que desarrolló, para efectos de construir su código en C ? ¿Requirieron ajustes?

Estas preguntas se repetirán para cada lenguaje que vaya explorando. Las respuestas a estas preguntas serán útiles para un comparativo de lenguajes al finalizar la lectura del libro.

www.detodoprogramacion.com

Notas:

Examen rápido

Parte 1 : Marque co n una cruz el inciso que considere que responde a la pregunta.

1) Es el marco de trabajo necesario para trabajar con programas desarrollados en C
(a) .NET Frame or
(b) J2SE
(c) No se requiere un marco de trabajo

2) El compilador de C es
(a) Bcc32
(b) Vbc.exe
(c) Javac

3) La biblioteca de clases que utiliza C se llama
(a) JFC
(b) BCL
(c) No tiene nombre

4) Al compilar un programa en C se genera un archivo de extensión
(a) E E
(b) Class
(c) OBJ

5) Al compilar un programa en C el archivo que se genera es llamado
(a) assembly
(b) bytecode
(c) machine code

Parte 2: Coloque en la línea la letra «F» si la sentencia es falsa, y «V» si la sentencia es verdadera.

_____ **6)** El compilador de C es en dos fases

_____ **7)** C soporta bucles de tipo Until.

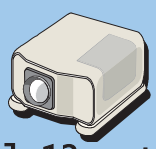
_____ **8)** NetBeans es una herramienta de desarrollo asociada a C

_____ **9)** La biblioteca de clase de C se compone de archivos de extensión .H

_____ **10)** En C para tener acceso a la biblioteca de clases es necesario referir el nombre físico de la biblioteca, dentro del programa, usando using.

12

Implementación de algoritmos en C++



lp12.ppt

Objetivos: Aprenderá a implementar algoritmos utilizando el lenguaje de programación C++.

1. Aprenderá cuáles son los requerimientos de ambiente para desarrollar en C++.
2. Aprenderá las generalidades del lenguaje C++, así como la estructura básica de sus programas.
3. Aprenderá a realizar cambios de estado, peticiones de datos, muestra de datos, condicionales y bucles usando C++.
4. Aprenderá a compilar y ejecutar programas desarrollados en C++.

www.detodoprogramacion.com

Contenido

Implementación de algoritmos en C++ 336
Ambiente de desarrollo 336
Estructura básica de un programa 339
Características generales del lenguaje 339
Tipos de datos 340
Declaración de variables y arreglos 340
Operadores principales 341
Cambios de estado 342
Mostrado de datos 342
Peticiones de datos 343
Estructuras de decisión 344
Estructuras de control 346
Compilación y ejecución de programas 348
Codificación de C1Mensaje en C++ 349
Codificación de C2Niveles en C++ 350
Codificación de C3Tabla en C++ 352
Codificación de C4MultiTabla en C++ 355
Codificación de C5Promedios en C++ 357
Desarrollando un programa de regla de tres simple usando C++ 360
Desarrollando un programa para el cálculo del área de un polígono irregular usando C++ 361
Desarrollando un programa con bucles, condicionales y arreglos usando C++ 362
Mapa mental del capítulo 363
Terminología 364
Preguntas 364
Examen rápido 365

Implementación de algoritmos en C++

C es un lenguaje de propósito general orientado a objetos, que permite desarrollar aplicaciones de código nativo.

Ambiente de desarrollo

El ambiente de desarrollo del lenguaje C se compone como sigue.

- *arco de trabajo.*

  C es un lenguaje que utiliza compilación en dos fases, es decir, se produce código máquina a partir del código fuente. C , a diferencia de Visual Basic.NET, C y Java, no produce código intermedio, por lo que no requiere de un marco de trabajo para que sus programas puedan funcionar. Esto

tiene la ventaja de no requerir instalaciones de productos para que el programa pueda funcionar en un entorno operativo determinado.

Las pruebas de este libro se hicieron con Borland C Compiler 5.5, que puede descargarse de forma gratuita en www.borland.com. Si es su caso, la ruta de herramientas del marco de trabajo puede agregarse desde línea de comandos, ejecutando lo siguiente:

```
PATH=”%PATH%”;”C:\Borland\Bcc55”
PATH=”%PATH%”;”C:\Borland\Bcc55\include”
PATH=”%PATH%”;”C:\Borland\Bcc55\bin”
```

Esto garantiza que el producto y su configuración, la biblioteca de clases, y las herramientas de línea de comando están disponibles.

- *Editor de programas.*

  Aunque se pueden editar programas C en cualquier editor de texto plano, podemos elegir herramientas de desarrollo más sofisticadas, tales como Borland Developer Studio, Borland C Builder, Microsoft Visual C , el mismo Microsoft Visual Studio.NET, o alternativas libres como Dev C .

- *ompilador de programas.*

  Por ser un lenguaje muy portable, C tiene muchos compiladores. El compilador de programas en C que utilizaremos en el libro es el programa BCC32.EXE, que viene incluido en Borland C Compiler 5.5, que puede ser descargado de la página eb de Borland de forma gratuita. Al compilar un programa en C se genera un archivo objeto (.OBJ), que es la representación del programa libre de ambig edades y errores de sintaxis. Este archivo servirá a la herramienta ILINK32.EXE, llamado también *enlazador*, que se encargará de producir el código máquina para la plataforma, uniendo los recursos de nuestro programa con los de la biblioteca de clases de C .

- *iblioteca de clases.*

  En el caso de C la biblioteca de clases se instala en el directorio \include del producto. Las funciones disponibles para el lenguaje se encuentran en bibliotecas de clases con la extensión .H, mismas que deberán ser referidas bajo su nombre físico desde los programas. La forma de hacer esto es la siguiente.

```
#include <iostream.h>
```

Al incluir una biblioteca en un programa podemos utilizar su funcionalidad. Esta funcionalidad está organizada internamente en las bibliotecas como

www.detodoprogramacion.com

una jerarquía de clases. Por ejemplo, al incluir iostream.h podremos invocar las instrucciones de la jerarquía sdt (standard), donde encontramos funciones de entrada y salida a consola, por ejemplo std::cin y std::cout. Para invocar dichas funciones es necesario mencionar la jerarquía a la que pertenecen.

Como es un poco molesto tener que estar repitiendo como prefijo la jerarquía de las funciones (std::), podemos utilizar un espacio de nombre al inicio el programa, después del #include.

Un espacio de nombres es un medio para la referencia lógica de la funcionalidad contenida en una biblioteca de clases, permitiendo el acceso simplificado a las clases. En C podemos establecer espacios de nombre escribiendo después de las líneas #include, líneas similares a esta.

```
using namespace std;
```

Con lo cual las instrucciones std::cin y std::cout podrían escribirse simplemente como cin y cout. Un espacio de nombres nos permite obviar la especificación de la jerarquía de clase a la que pertenecen las funciones.

Figura 12.01

Proceso de compilación y ejecución en C++

Estructura básica de un programa

La estructura base de todo programa en el lenguaje C es la siguiente.

```
1  #include <iostream.h>
2  using namespace std;
3
4  void main()
5  {
6
7     ...
8
   }
```

El procedimiento void le indica al compilador que nuestro procedimiento no retorna valores en forma de función, es decir, se trata de un programa completo. Nuestro procedimiento principal deberá incluirse en el espacio que ocupa la línea 7 de la ilustración.

Características generales del lenguaje

A C le aplican las siguientes reglas generales básicas.

1. Es sensible a mayúsculas y minúsculas (es *case sensitive*), por lo que el uso de mayúsculas y minúsculas son relevantes para el compilador.
2. Los bloques de código se encierran entre llaves “ ”.
   a. Las llaves van siempre por pares.
   b. Todas las llaves que se abran deben cerrarse.
   c. La última llave en abrirse, será siempre la primera en cerrarse.
3. Se considera un bloque de código la presencia de dos o más líneas de código.
4. Las líneas en C se terminan con punto y coma “;”.
. Las palabras reservadas, intrínsecas del lenguaje C , se escriben en minúsculas.
6. Los comentarios en C se especifican colocando doble diagonal “ ” al inicio de una línea.
. Para utilizar variables, antes deben declararse.

Tipos de datos

Los tipos de datos básicos del lenguaje C son los siguientes.

Tipo	Consumo en Bytes	Dominio de tipo
bool		r e o alse
short	2 bytes	32,767 a 32,767
double	8 bytes	De 10^{308} a 10^{308}
int	4 bytes	2.147.483.648 a 2.147.483.647
string	Clase (1 byte por caracter, aprox)	Arreglo de caracteres.

Declaración de variables y arreglos

La forma de declarar variables en el lenguaje C es utilizando la siguiente sintaxis.

tipo ariable [= *E presión*];

Donde *ariable* es un nombre de variable válido, y *tipo* es un tipo de datos reconocido por el lenguaje. En C se puede asignar un valor de inicio al momento de declarar una variable, asignando una *E presión*, con posterioridad al nombre de la variable.

A continuación unos ejemplos.

```
int i;
int i=1;
string Mensaje;
```

Equivalencias

Miniespecificación	Diagrama de flujo	C++
(i,i[1-n])		int i=1;
(Suma,d[1-n])		decimal Suma=0;

Los arreglos se definen en el lenguaje C de la forma siguiente.

```
tipo  ariable Elementos ;
```

Donde *ariable* es un nombre de variable válido, *tipo* es un tipo de datos reconocido por el lenguaje, y *Elementos* es el número de elementos del arreglo. En C los arreglos son de base cero, es decir, que los subíndices son 0, 1, 2, 3, y así. Declarar un arreglo como Numeros(3), genera un arreglo de 3 elementos con los subíndices 0,1,2 .

A continuación unos ejemplos.

```
int Ventas [3];
int[ ] Ventas [ ] {20000,15000};
```

Operadores principales

Los operadores principales del lenguaje C , aplicables a los ejercicios tratados en el libro, son los siguientes.

Tipo	Operador	Nombre
Asignación	=	Igual
Aritmético	+	Suma/ concatenación de cadenas
	-	Resta
	*	Multiplicación
	/	División
	^	Exponenciación
Lógico		Disyuntiva
	||	Conjuntiva
	!	Negación
Comparativos	>	Mayor que
	<	Menor que
	>=	Mayor o igual que
	<=	Menor o igual que
	==	Igual a
	<>	Distinto a

Asignación incluyente	+=	Suma incluyente
	-=	Resta incluyente

Cambios de estado

Los cambios de estado en el lenguaje C se realizan utilizando el operador de asignación (=).

```
ariable = E presión
```

Donde *ariable* es el nombre de una variable previamente declarada y *E presión* es el valor que se le asigna a la variable. Estos son unos ejemplos.

```
AreaCirculo = Lado^2;
i = i + 1;
i += 1;
```

En el ejemplo, las últimas dos líneas hacen lo mismo.

Equivalencias

Miniespecificación	Diagrama de flujo	C++
Promedio ← Suma/5	Promedio = Suma / 5	Promedio = Suma/5;
Suma ← Suma + Entrada(i)	Suma = Suma + Entrada(i)	Suma += Entrada(i);
i ← i + 1	i = i + 1	i += 1;

Mostrado de datos

El mostrado de datos se realiza en el lenguaje de C a través de la orden cout.

```
cout << E presión [ << E presión1 [<< endl] ];
```

Donde *E presión* es la expresión a mostrar. La salida en C es tratada como un *flujo de datos* (*stream*) que no termina hasta que se concluye la línea (;). Dentro del mismo flujo se puede unir otra expresión (*E presión1*) o más, utilizando el símbolo << como continuador de flujo entre las expresiones. También podemos agregar la constante endl, que indica el salto de línea.

Ejemplos.

```
cout << "Dame un número: ";
cout << "Suma: " << Suma;
cout << "Suma: " << Suma;
```

Equivalencias

Miniespecificación
« "Suma: " + Suma
Diagrama de flujo
C++
`cout << "Suma:" << Suma << endl;`

Peticiones de datos

Las peticiones de datos se realizan en el lenguaje de C a través de la orden cin. Su sintaxis es la siguiente.

```
cin >> ariable;
```

Donde *ariable* es la variable en la que se almacenará lo que el usuario introduzca a la interfaz.

Generalmente las peticiones de datos se realizan a solicitud, por lo que es común asociar una petición con un mostrado de datos previo, quedando como sigue.

```
cout << "E  presión";
cin >>   ariable;
```

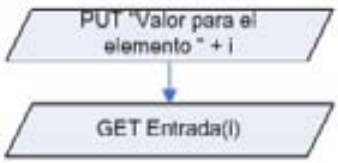
Equivalencias

Miniespecificación

« "Valor para el elemento " + i

» Entrada(i)

Diagrama de flujo



C++

```
cout << "Valor para el elemento " << i << ":" << endl;
cin >> Entrada[i];
```

Estructuras de decisión

Las estructuras de decisión se realizan en el lenguaje C a través de la estructura if.

```
If  ondición
{
        nstrucciones por verdadero
[   } else {
        nstrucciones por falso ]
}
```

Donde *ondición* es la expresión lógica a evaluar. En caso de que *ondición* sea True, se ejecutará el bloque de código que contiene las *nstrucciones por verdadero*. De manera opcional se puede especificar un bloque de código a ejecutar si *ondición* resuelve por falso, utilizando else. En ese caso, se ejecutarían las *nstrucciones por falso*. La estructura if siempre debe terminar.

Ejemplos.

```
if Monto > LimiteCredito
{
        Venta = “Rechazada”;
}
else
{
        Venta = “Aprobada”;
}
```

Equivalencias

Miniespecificación

```
Si Entrada(i) >= Promedio Entonces
        « “Numero ” + Entrada(i) + “ mayor o igual al promedio”
De lo contrario
        « “Numero ” + Entrada(i) + “ menor al promedio”
Fin Si
```

Diagrama de flujo

C++

```
if  Entrada(i) >= Promedio
{
  cout << “Numero ” << Entrada(i) << “ mayor o igual al promedio” << endl;
}
else
{
    cout << “Numero ” << Entrada(i) << “ menor al promedio” << endl;
}
```

Estructuras de control

Las estructuras de control se realizan en el lenguaje C a través de la estructura do while.

Es importante señalar la forma en que se manejan las estructuras de control con condición al final en C , dado que las miniespecificaciones que pretendemos codificar están diseñadas de esa forma.

Para empezar, una estructura de control con condición al final representa una estructura de control de tipo until (hasta que). Dado que C no posee estructura until (hasta que), se deberá especificar una condición que mantenga el proceso ejecutándose, mientras la condición se esté cumpliendo.

Supongamos que tenemos un bucle del 1 al 10. En las estructuras until la condición provocaba la terminación del ciclo, de ahí que el proceso se mantenía ejecutando hasta que 1 > 10. Con while la condición tendría que cambiarse a 1 <=10, es decir, mientras sea menor o igual a 10.

```
do
{
        nstrucciones

while (  ondición);
```

Donde *nstrucciones* es el bloque de código que se estará repitiendo hasta que *ondición* se cumpla. Todo lo que esté entre do y while se considera código controlado. Es importante no olvidar colocar y cambiar el estado de la variable de trabajo, dentro del código de control.

Ejemplos.

```
i = 1;
do
{
        Console.WriteLine("Valor del contador: " + i);
        i += 1;
}
while (i <= 5);
```

Equivalencias

Miniespecificación

i ← 1

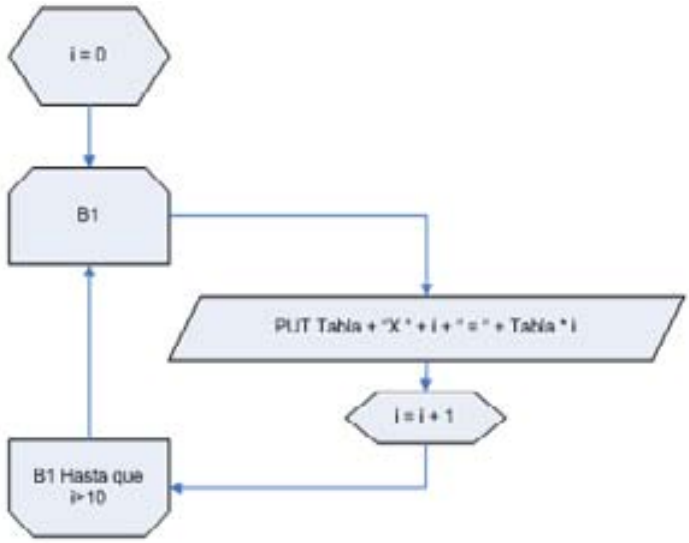
Inicio

« Tabla + “X” + i + “=” + Tabla*i

i ← i + 1

Hasta que i > 10

Diagrama de flujo



C++

```
i = 1;
do
{
        cout << Tabla << “X” << i << “=” << Tabla*i << endl;
        i += 1;
}
while (i <= 10);
```

Vea cómo la condición del bucle cambia a la condición opuesta, dado que cambiamos de un enfoque until a un enfoque while.

Aunque no manejamos estructuras de control con condición al inicio en este libro, proporcionamos la sintaxis de su implementación en C .

```
while ( ondición)
{
        nstrucciones
```

Donde *nstrucciones* es el bloque de código que se estará repitiendo mientras que *ondición* se cumpla por verdadero.

Compilación y ejecución de programas

La compilación de los programas se realiza utilizando el compilador de línea de comandos de C , llamado BCC32.EXE. La sintaxis básica de compilación es la siguiente.

```
bcc32 Programa.cpp
```

Donde *Programa* es el programa a compilar.

El comando se ejecuta desde la línea de comandos. Es importante no omitir la extensión de los programas desarrollados en C (.cpp), dado que la compilación generaría error.

NOTA

En ocasiones las bibliotecas y herramientas no se localizan, aún y cuando las rutas de los recursos ya han sido agregadas a la variable de ambiente PATH. En ese caso, compile de la siguiente manera:

```
bcc32.exe –I"c:\borland\bcc55\include" –L"c:\borland\bcc55\lib" Programa.cpp
```

Al concluir la compilación sin errores, se habrá creado un programa de extensión .obj, que se trata de código precompilado de C . El mismo compilador intenta el enlazado del programa con todos los recursos que consume, al lanzar de forma automática una instancia de ILINK32.EXE. Al finalizar el proceso, se habrá generado un archivo de extensión .exe. Este archivo es código máquina ejecutable por la plataforma en que se compiló el programa.

El programa podrá ejecutarse desde la línea de comandos, invocando el programa ejecutable que se generó.

Programa

Donde *Programa* es el nombre del archivo .exe que se desea ejecutar.

Ejercicio 12.01

Codificación de C1Mensaje en C++

En el presente ejercicio se codificará el primer caso de estudio, utilizando el lenguaje C .

1. Utilice un editor de texto plano (*loc de notas*, por ejemplo), para escribir el código fuente llamado C1Mensaje.cpp; el archivo deberá guardarse en el directorio de trabajo del libro (C:\APIP).

Este programa tiene como finalidad comprobar técnicas y herramientas de desarrollo. Lo que el programa debe hacer es preguntar por un mensaje de saludo, para posteriormente mostrar el mensaje que se le proporcionó.

Miniespecificación

```
1   INICIO:
2           (Mensaje, x(30))
3           « "Escribe un mensaje a mostrar: "
4           » Mensaje
5           « Mensaje
6   :FIN
```

Codificación de **C1Mensaje.cpp**

```
1   #include <iostream.h>
2   using namespace std;
3
4   void main()
5   {
6
7           string Mensaje;
8           cout << "Escribe un mensaje a mostrar: ";
9           cin >> Mensaje;
10          cout <<  Mensaje << endl;
```

```
11
12 }
```

2. Compile el programa utilizando la siguiente línea.

```
bcc32 C1Mensaje.cpp
```

3. Si no genera errores, ejecute el programa de la siguiente forma.

```
C1Mensaje.exe
```

Si todo ocurrió como se esperaba, la salida será la siguiente:

Entrada	Escribe un mensaje a mostrar: Aprenda
Salida	Aprenda

FIN DEL EJERCICIO ✱

Ejercicio 12.02

Codificación de C2Niveles en C++

En el presente ejercicio se codificará el segundo caso de estudio, utilizando el lenguaje C .

1. Utilice un editor de texto plano (*loc de notas*, por ejemplo), para escribir el código fuente llamado C2Niveles.cpp; el archivo deberá guardarse en el directorio de trabajo del libro (C:\APIP).

Se desea saber la disponibilidad del servicio de Internet para una persona, expresado en horas de disponibilidad.

Dicho nivel de servicio depende del grado de disponibilidad que ofrece la computadora que utiliza, así como del grado de disponibilidad del enlace a Internet que utiliza la máquina.

El horario de labores de la persona que utiliza la computadora y el enlace a Internet es de 8 horas.

Para poder realizar los cálculos adecuadamente, tome en cuenta que la disponibilidad de un sistema de dispositivos (dos o más dispositivos que trabajan de forma conjunta) se calcula multiplicando los porcentajes de

grado de disponibilidad de cada uno de ellos, y el resultado siempre será menor a la disponibilidad del menor.

Para calcular el tiempo de disponibilidad de un servicio (en este caso el acceso a la información publicada en Internet), sólo basta con multiplicar el tiempo en que es requerido el servicio por la disponibilidad del sistema de dispositivos.

Miniespecificación

```
1   INICIO:
2           (DC,d[0-1])
3           (DE,d[0-1])
4           (TL,d[0-1])
5           (DS,d[0-1])
6           (DH,d[0-1])
7           « "Grado de disponibilidad de la computadora:"
8           » DC
9           « "Grado de disponibilidad del enlace:"
10          » DE
11          « "Tiempo laborable en horas:"
12          » TL
13          DS ← DC * DE
14          DH ← DS * TL
15          « "La disponibilidad del servicio de Internet es de " + DH + " horas"
16  :FIN
```

Codificación de **C2Niveles.cpp**

```
1   #include <iostream>
2   using namespace std;
3
4   void main()
5   {
6
7           double DC;
8           double DE;
9           double TL;
10          double DS;
11          double DH;
12
13          cout << "Grado de disponibilidad de la computadora: ";
14          cin >> DC;
15          cout << "Grado de disponibilidad del enlace: ";
16          cin >> DE;
17          cout << "Tiempo laborable en horas:  ";
18          cin >> TL;
19
20          DS= DC*DE;
```

```
21          DH= DS*TL;
22
23          cout << "La disponibilidad del servicio de internet es de "<< DH << " horas" << endl;
24
25  }
```

2. Es importante analizar que al preguntar los datos compile el programa utilizando la siguiente línea.

```
bcc32 C2Niveles.cpp
```

3. Si no genera errores, ejecute el programa de la siguiente forma.

```
C2Niveles.exe
```

Si todo ocurrió como se esperaba, la salida será la siguiente:

Entrada	Grado de disponibilidad de la computadora: .98
Entrada	Grado de disponibilidad del enlace: .85
Entrada	Tiempo laborable en horas: 8
Salida	La disponibilidad del servicio de internet es de 6.6640 horas

FIN DEL EJERCICIO ✱

Ejercicio 12.03

Codificación de C3Tabla en C++

En el presente ejercicio se codificará el tercer caso de estudio, utilizando el lenguaje C .

1. Utilice un editor de texto plano (*loc de notas*, por ejemplo), para escribir el código fuente llamado C3Tabla.cpp; el archivo deberá guardarse en el directorio de trabajo del libro (C:\APIP).

El programa debe mostrar cualquier tabla de multiplicar que le solicitemos. La única restricción es que la tabla deberá ser de un número mayor o igual a 1, y menor o igual a 10.

Si se pide una tabla de un número que no esté entre el 1 y el 10, el programa debe indicar, mediante un mensaje, que no puede continuar.

Miniespecificación

```
1   INICIO:
2           (i,i[0-n])
3           (Tabla,i[1-10])
4           « "Qué tabla deseas: "
5           » Tabla
6           Si Tabla >= 1 ∧ Tabla <=10 Entonces
7                   i ← 1
8                   Inicio
9                           « Tabla + " X" + i + "=" + Tabla * i
10                          i ← i + 1
11                  Hasta que i > 10
12          De lo contrario
13                  « "Válido sólo del 1 al 10"
14          Fin Si
15  :FIN
```

Codificación de **C3Tabla.cpp**

```
1   #include <iostream.h>
2   using namespace std;
3
4   int main()
5   {
6           int Tabla;
7           int i;
8
9           cout << "Qué tabla deseas: ";
10          cin >> Tabla;
11
12          if (Tabla >=1 & Tabla<=10 )
13          {
14                  i=1;
15                  do
16                  {
17                          cout << Tabla << " X "<< i << " = "<< Tabla*i << endl;
18                          i+=1;
19                  }
20                  while (i <= 10);
21          }
22          else
23          {
24                  cout << "Valido solo del 1 al 10" << endl;
25          }
26  }
```

2. Compile el programa utilizando la siguiente línea.

```
bcc32 C3Tabla.cpp
```

3. Si no genera errores, ejecute el programa de la siguiente forma.

```
C3Tabla.exe
```

Si todo ocurrió como se esperaba, la salida será la siguiente:

Entrada	Que tabla deseas: 5
Salida	5 X 1 = 5
	5 X 2 = 10
	5 X 3 = 15
	5 X 4 = 20
	5 X 5 = 25
	5 X 6 = 30
	5 X 7 = 35
	5 X 8 = 40
	5 X 9 = 45
	5 X 10 = 50

Entrada	Que tabla deseas: 25
Salida	Válido sólo del 1 al 10

FIN DEL EJERCICIO ✱

Ejercicio 12.04

Codificación de C4MultiTabla en C++

En el presente ejercicio se codificará el cuarto caso de estudio, utilizando el lenguaje C .

1. Utilice un editor de texto plano (*loc de notas*, por ejemplo), para escribir el código fuente llamado C4MultiTabla.cpp; el archivo deberá guardarse en el directorio de trabajo del libro (C:\APIP).

El programa debe mostrar todas las tablas de multiplicar del 1 al 5, incluyendo un encabezado por tabla.

Miniespecificación

```
1   INICIO:
2           (i,i[0-n])
3           (y,i[0-n])
4           i ← 1
5           Inicio
6                   « "TABLA: " + i
7                   y ← 1
8                   Inicio
9                           « i + " X" + y + "=" + i * y
10                          y ← y + 1
11                  Hasta que y > 10
12                  i ← i + 1
13          Hasta que i > 5
14  :FIN
```

Codificación de **C4MultiTabla.cpp**

```
1   #include <iostream.h>
2   using namespace std;
3
4   int main()
5   {
6
7       int i;
8           int y;
9
10          i=1;
11          do
```

```
12          {
13                  cout << "Tabla: "<< i << endl;
14                  y=1;
15                  do
16                  {
17                          cout << i << " X "<< y << " = "<< i*y << endl;
18                          y += 1;
19                  }
20                  while (y <= 10);
21                  i += 1;
22          }
23          while (i <= 5);
24  }
```

2. Compile el programa utilizando la siguiente línea.

```
bcc32 C4MultiTabla.cpp
```

3. Si no genera errores, ejecute el programa de la siguiente forma.

```
C4MultiTabla.exe
```

Si todo ocurrió como se esperaba, la salida será la siguiente:

Salida

```
Tabla: 1
1 X 1 = 1
1 X 2 = 2
1 X 3 = 3
1 X 4 = 4
1 X 5 = 5
1 X 6 = 6
1 X 7 = 7
1 X 8 = 8
1 X 9 = 9
1 X 10 = 10
Tabla: 2
2 X 1 = 2
2 X 2 = 4
2 X 3 = 6
2 X 4 = 8
```

```
...

5 X 7 = 35
5 X 8 = 40
5 X 9 = 45
5 X 10 = 50
```

FIN DEL EJERCICIO ✱

Ejercicio 12.05

Codificación de C5Promedios en C++

En el presente ejercicio se codificará el quinto caso de estudio, utilizando el lenguaje C .

1. Utilice un editor de texto plano (*loc de notas*, por ejemplo), para escribir el código fuente llamado C5Promedios.cpp; el archivo deberá guardarse en el directorio de trabajo del libro (C:\APIP).

El programa debe pedir 5 números, para después mostrarlos indicando si están o no por encima de su promedio. Al final deberá enviarse un resumen en donde se indique la suma total de los números, y el promedio calculado.

Miniespecificación

```
1   INICIO:
2           (i,i[0-n])
3           (Entrada(5),i[0-n])
4           (Suma,d[0-n])
5           (Promedio,d[0-n])
6           Suma ← 0
7           i ← 1
8           Inicio
9                   « "Valor para el elemento " + i + ": "
10                  » Entrada(i)
11                  Suma ← Suma + Entrada(i)
12                  i ← i + 1
13          Hasta que i > 5
14          Promedio ← Suma / 5
15          i ← 1
16          Inicio
```

```
17                      Si Entrada(i) >= Promedio Entonces
18                          « "Número " + Entrada(i) + " mayor o igual al promedio: "
19                      De lo contrario
20                          « "Número " + Entrada(i) + " menor al promedio: "
21                      Fin Si
22                      i ← i + 1
23              Hasta que i > 5
24              « "Suma: " + Suma
25              « "Promedio: " + Promedio
26      :FIN
```

Codificación de **C5Promedios.cpp**

```
1   #include <iostream.h>
2   using namespace std;
3
4   int main()
5   {
6
7           int i;
8           int Entrada[5];
9           int Suma;
10          float Promedio;
11
12          Suma=0;
13
14          i=1;
15          do
16          {
17                  cout << "Valor para el elmento " << i << ": " ;
18                  cin >> Entrada[i-1];
19                  Suma+=Entrada[i-1];
20                  i += 1;
21          }
22          while (i <= 5);
23
24          Promedio = Suma / 5;
25
26          i=1;
27          do
28          {
29                  if (Entrada[i-1] >= Promedio)
30                  {
31                    cout << "Numero " << Entrada[i-1] << " mayor igual al promedio" << endl;
32                  }
33                  else
34                  {
35                     cout << "Numero " << Entrada[i-1] << " menor al promedio" << endl;
36                  }
37                  i += 1;
```

```
38          }
39          while (i <= 5);
40
41          cout << "Suma: " << Suma << endl;
42          cout << "Promedio: " << Promedio << endl;
43
44  }
45
46
```

2. Es importante mencionar que este programa tiene discrepancias con la miniespecificación, provocadas por el cambio de enfoque en las estructuras de control. El hecho de cambiar las estructuras de tipo until, por estructuras de tipo while, hace que las condiciones que eran i > 5 en la miniespecificación (líneas 13 y 23) pasen a ser i<=5 en el programa (líneas 22 y 39).

3. En las llamadas al arreglo, en la miniespecificación se coloca como subíndice i (líneas 11, 17, 18 y 20), mientras que en el programa se refiere al subíndice como i-1 (líneas 18, 19, 29, 31, 35). Esto se origina porque para la miniespecificación el primer elemento del arreglo tiene el subíndice 1, pero para el programa el primer elemento tiene el subíndice 0; para la miniespecificación el segundo elemento tiene el subíndice 2, y para el programa es 1. Si se observa, el subíndice del programa es igual al subíndice de la miniespecificación, menos 1. Al colocar en el programa el subíndice como i-1 homologamos el funcionamiento de la miniespecificación y del programa, y de esa forma el resto de la miniespecificación sigue funcionando sin cambio, además que el usuario no se percata del cambio.

4. Compile el programa utilizando la siguiente línea.

```
bcc32 C5Promedios.cpp
```

5. Si no genera errores, ejecute el programa de la siguiente forma.

```
C5Promedios.exe
```

Si todo ocurrió como se esperaba, la salida será la siguiente:

Salida	Valor para el elemento 1: 1 Valor para el elemento 2: 2 Valor para el elemento 3: 3

```
Valor para el elemento 4: 4
Valor para el elemento 5: 5
Número 1 es menor al promedio
Número 2 es menor al promedio
Número 3 es mayor o igual al promedio
Número 4 es mayor o igual al promedio
Número 5 es mayor o igual al promedio
Suma: 15
Promedio: 3
```

FIN DEL EJERCICIO ✱

Ejercicio 12.06

Desarrollando un programa de regla de tres simple usando C++

Utilizando C , realice las siguientes actividades:

- ❒ Desarrolle el código fuente.
- ❒ Asigne como nombre del programa TresSimple.cpp .
- ❒ Compílelo.
- ❒ Ejecútelo.

Elabore un programa que realice el cálculo de una regla de tres simple. Como sabe, dicha regla permite calcular la proporción de un número respecto a otro que actúa como referencia de la proporción. Este programa contesta preguntas del tipo ¿qué porcentaje de 200 es 135? El 200 sería el número base de la proporción, mientras que el 135 sería la proporción. La solución se calcula multiplicando la proporción por 100, y dividiéndolo entre el número base de la proporción.

El programa deberá mostrar el resultado del cálculo.

FIN DEL EJERCICIO ✱

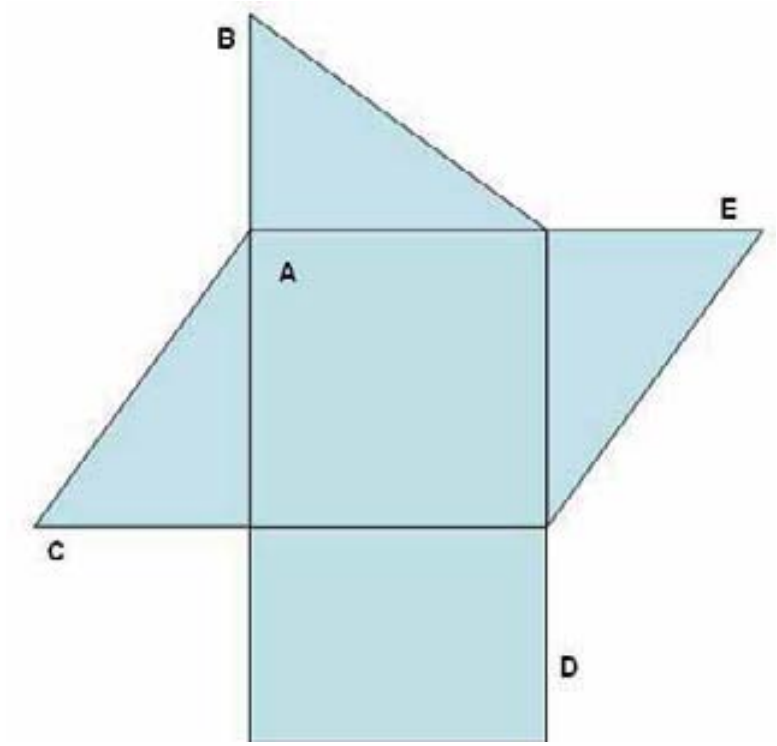
Ejercicio 12.07

Desarrollando un programa para el cálculo del área de un polígono irregular usando C++

Utilizando C , realice las siguientes actividades:

- ❒ Desarrolle el código fuente.
- ❒ Asigne como nombre del programa Poligono.cpp .
- ❒ Compílelo.
- ❒ Ejecútelo.

Observe la siguiente figura.



El programa debe calcular el área del polígono. La figura se compone de un cuadrado perfecto (A), tres triángulos rectángulos iguales (B, C, E), cuya base mide lo mismo que uno de los lados del cuadrado, y un rectángulo cuyo ancho mide lo mismo que uno de los lados del cuadrado, mientras que la altura mide lo mismo que la altura de los triángulos. Se pide que haga un programa que calcule el área total.

Por cultura general, sabemos que el área de un rectángulo se calcula multiplicando su base por su altura; el área de un triángulo rectángulo se calcula multiplicando su base por su altura, y luego dividiendo el resultado entre dos; finalmente, el área de un cuadrado se calcula elevando al cuadrado la medida de uno de sus lados.

FIN DEL EJERCICIO ✱

Ejercicio 12.08

Desarrollando un programa con bucles, condicionales y arreglos usando C++

Utilizando C , realice las siguientes actividades:

- ❒ Desarrolle el código fuente.
- ❒ Asigne como nombre del programa Descuento.cpp .
- ❒ Compílelo.
- ❒ Ejecútelo.

Una tienda departamental ofrece descuentos directos sobre las compras que los clientes realizan.

El descuento normal va en función al tipo de cliente que la persona sea: A los clientes normales se les ofrece un 10 de descuento, y a los clientes premier un 15 de descuento.

Actualmente se tiene una promoción: en la compra de 5 artículos, se ofrecerá un descuento adicional al normal. Si el monto de la compra no excede de 1,000 pesos, se agrega un 5 de descuento; si el monto de la compra está entre 1,000 y 2,000 pesos, se agrega un descuento del 10 ; si la compra es mayor a 2,000 pesos, entonces el descuento adicional será del 15 .

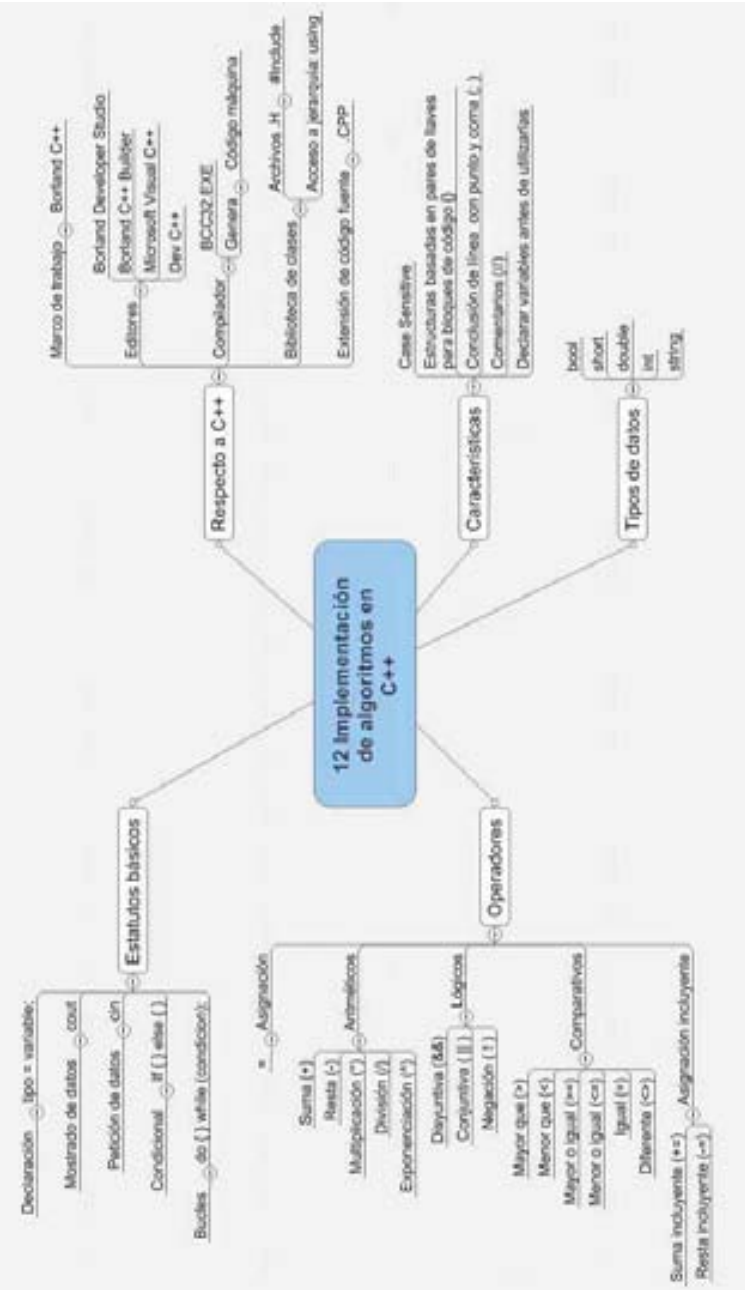
En ese sentido, el menor descuento que se puede obtener es cuando un cliente normal compra menos de 1,000 pesos (15), y el máximo descuento que se puede obtener es cuando un cliente premier compra más de 2,000 pesos (30).

El programa deberá pedir la información de los precios de 5 productos, y determinar el monto de descuento aplicable.

Utilice arreglos.

FIN DEL EJERCICIO ✱

Mapa mental del capítulo



Terminología

.H, 337
.OBJ, 337
BCC32.EXE, 337
Borland C++ Builder, 337
Borland C++ Compiler 5.5, 337
Borland Developer Studio, 337
Dev-C++, 337
Enlazador, 337
Flujo de datos, 343
ILINK32.EXE, 337
Microsoft Visual C++, 337
Stream, 343

Preguntas

12.1 Tome tiempo de los siguientes aspectos:

12.1.1 ¿Tiempo en que se desarrollaron los casos 6, 7 y 8 en C ?

12.1.2 ¿Veces en que fue necesario corregir cada uno de los programas 6, 7 y 8, a fin de corregir el programa por errores de sintaxis?

12.1.3 ¿Qué tan fácil fue desarrollar los casos 6, 7 y 8 en C , en su opinión?

12.1.4 ¿Qué tan útiles fueron las miniespecificaciones que desarrolló, para efectos de construir su código en C ? ¿Requirieron ajustes?

Estas preguntas se repetirán para cada lenguaje que vaya explorando. Las respuestas a estas preguntas serán útiles para un comparativo de lenguajes al finalizar la lectura del libro.

Examen rápido

Parte 1 : Marque co n una cruz el inciso que considere que responde a la pregunta.

1) Es el marco de trabajo necesario para trabajar con programas desarrollados en C
(a) .NET Frame or
(b) J2SE
(c) No se requiere un marco de trabajo

2) El compilador de C es
(a) Bcc32
(b) Vbc.exe
(c) Javac

3) La biblioteca de clases que utiliza C se llama
(a) JFC
(b) BCL
(c) No tiene nombre

4) Al compilar un programa en C se genera un archivo de extensión
(a) E E
(b) Class
(c) OBJ

5) Al compilar un programa en C el archivo que se genera es llamado
(a) assembly
(b) bytecode
(c) machine code

Parte 2: Coloque en la línea la letra «F» si la sentencia es falsa, y «V» si la sentencia es verdadera.

6) El compilador de C es en dos fases. _____

7) C soporta bucles de tipo Until. _____

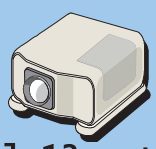
8) NetBeans es una herramienta de desarrollo asociada a C _____

9) La biblioteca de clase de C se compone de archivos de extensión .H _____

10) En C para tener acceso a la biblioteca de clases es necesario referir el nombre físico de la biblioteca, dentro del programa, usando Imports. _____

13

Implementación de algoritmos en Java



lp13.ppt

Objetivos: Aprenderá a implementar algoritmos utilizando el lenguaje de programación Java.

1. Aprenderá cuáles son los requerimientos de ambiente para desarrollar en Java.
2. Aprenderá las generalidades del lenguaje Java, así como la estructura básica de sus programas.
3. Aprenderá a realizar cambios de estado, peticiones de datos, muestra de datos, condicionales y bucles usando Java.
4. Aprenderá a compilar y ejecutar programas desarrollados en Java.

www.detodoprogramacion.com

Contenido

Implementación de algoritmos en Java 368
Ambiente de desarrollo 368
Estructura básica de un programa 371
Características generales del lenguaje 371
Tipos de datos 372
Declaración de variables y arreglos 372
Operadores principales 373
Cambios de estado 374
Mostrado de datos 375
Peticiones de datos 375
Estructuras de decisión 377
Estructuras de control 379
Compilación y ejecución de programas 381
Codificación de C1Mensaje en Java 382
Codificación de C2Niveles en Java 383
Codificación de C3Tabla en Java 385
Codificación de C4MultiTabla en Java 388
Codificación de C5Promedios en Java 390
Desarrollando un programa de regla de tres simple usando Java 394
Desarrollando un programa para el cálculo del área de un polígono irregular usando Java 394
Desarrollando un programa con bucles, condicionales y arreglos usando Java 396
Mapa mental del capítulo 397
Terminología 398
Preguntas 398
Examen rápido 399

Implementación de algoritmos en Java

Java es un lenguaje de propósito general orientado a objetos, que permite desarrollar aplicaciones bajo la plataforma de desarrollo de *2 E (ava 2 Plataform tandard Edition*).

Ambiente de desarrollo

El ambiente de desarrollo del lenguaje Java se compone como sigue.

- *arco de trabajo.*

  Para el desarrollo de Java es necesario el marco de trabajo denominado (*ava evelopment it*), disponible de forma gratuita desde la página eb de Sun Microsystems (www.sun.com). Deberá instalar el ambiente para poder probar los programas en Java.

Las pruebas de este libro se hicieron con la versión Java 2 Plataform Standard Edition, Development Kit 5.0. Si es su caso, la ruta de herramientas del marco de trabajo puede agregarse desde línea de comandos, ejecutando lo siguiente:

```
PATH=”%PATH%”;”C:\Archivos de programa\Java\jdk1.5.0_06\bin”
PATH=”%PATH%”;”C:\Archivos de programa\Java\jdk1.5.0_06\include”
```

Esto garantiza que la biblioteca de clases, y las herramientas de línea de comando están disponibles.

- *Editor de programas.*

  Aunque se pueden editar programas Java en cualquier editor de texto plano, podemos elegir herramientas de desarrollo más sofisticadas, tales como Borland JBuilder, NetBeans, JCreator, y el mismo Microsoft Visual Studio.NET.

- *ompilador de programas.*

  Por ser un lenguaje muy portable, Java tiene muchos compiladores. El compilador de programas en Java que utilizaremos en el libro es el programa JAVAC.EXE, que viene incluido en JDK. Al compilar un programa en Java se genera un archivo de extensión .class.

  Los archivos .class generados por el compilador son código intermedio, a los que se les da el nombre de *bytecode*. Estos programas pueden ser ejecutados mediante un intérprete denominado JAVA.EXE, que pone al programa en contacto con el motor de ejecución de J2SE, denominado JVM (*ava irtual achine*).

- *iblioteca de clases.*

  En el caso de Java la biblioteca de clases se instala en el directorio \include del producto y recibe el nombre de JFC (*ava oundation lass*). Las funciones disponibles para el lenguaje se encuentran en archivos de extensión .H, que al instalar el ambiente de desarrollo quedan disponibles para su uso en los programas sin necesidad de hacer referencia al archivo físico.

  Las clases contenidas en los archivos de extensión .H se organizan a través de una jerarquía de clases, que en el caso de Java se conocen como *pa uetes* (*Pac ages*). Para tener acceso a la funcionalidad de una clase desde un programa es necesario especificar toda su jerarquía al momento de invocarla. Por ejemplo, si al querer manejar operaciones de entrada y salida deseamos invocar la clase que permite recibir un flujo de datos desde la consola, es necesario invocar la clase java.io.InputStreamReader(). java.io es la jerarquía del paquete, mientras que InputStreamReader() es la clase.

www.detodoprogramacion.com

Como es un poco molesto tener que estar repitiendo como prefijo la jerarquía del paquete (java.io) podemos predeterminar referencias a los paquetes de clases utilizando la instrucción import, a lo que se conoce como *importar pa uetes*. Para hacer esto podemos escribir al inicio del programa líneas similares a estas.

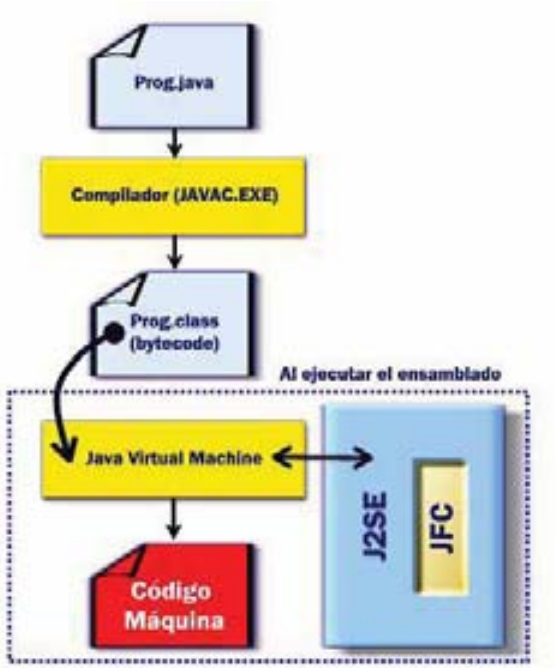
```
import java.io.*;
import java.lang.*;
```

Al aplicar las instrucciones anteriores, por ejemplo, la clase java.io.InputStreamReader() podría referirse simplemente como InputStreamReader(), dado que se importó java.io.

El uso de espacios de nombres nos permite obviar la especificación de la jerarquía de clases al momento de invocar la funcionalidad de las bibliotecas.

Figura 13.01

Proceso de compilación y ejecución en Java



Estructura básica de un programa

La estructura base de todo programa en el lenguaje Java es la siguiente.

```
1   import java.io.*;
2   import java.lang.*;
3
4   class   ombre  lase
5   {
6           public static void main(String args[])
7                   throws IOException
8     {
9
10                  ...
11
12          }
    }
```

Por convencionalismo se debe colocar como nombre de la clase el nombre del programa. Nuestro procedimiento principal deberá incluirse en el espacio que ocupa la línea 10 de la ilustracón.

Características generales del lenguaje

A Java le aplican las siguientes reglas generales básicas.

1. Es sensible a mayúsculas y minúsculas (es *case sensitive*), por lo que el uso de mayúsculas y minúsculas son relevantes para el compilador.
2. Los bloques de código se encierran entre llaves “ ”.
    a. Las llaves van siempre por pares.
    b. Todas las llaves que se abran deben cerrarse.
    c. La última llave en abrirse, será siempre la primera en cerrarse.
3. Se considera un bloque de código la presencia de dos o más líneas de código.
4. Las líneas en Java se terminan con punto y coma “;”.
. Las palabras reservadas, intrínsecas del lenguaje Java, se escriben en minúsculas.
6. Los comentarios en Java se especifican colocando doble diagonal “ ” al inicio de una línea.
. Para utilizar variables, antes deben declararse.

Tipos de datos

Los tipos de datos básicos del lenguaje Java son los siguientes.

Tipo	Consumo en Bytes	Dominio de tipo
boolean		r e o alse
byte	1 byte	128 a 127
long	8 bytes	9,223,372,036,854,775,808 a 9,223,372,036,854,775,807
int	4 bytes	2,147,483,648 a 2,147,483,647
String	Clase (1 byte por caracter, aprox)	Arreglo de caracteres.

Declaración de variables y arreglos

La forma de declarar variables en el lenguaje Java es utilizando la siguiente sintaxis.

tipo ariable [= *E presión*];

Donde *ariable* es un nombre de variable válido, y *tipo* es un tipo de datos reconocido por el lenguaje. En Java se puede asignar un valor de inicio al momento de declarar una variable, asignando una *E presión*, con posterioridad al nombre de la variable.

A continuación unos ejemplos.

```
int i;
int i=1;
String Mensaje;
```

Equivalencias

Miniespecificación	Diagrama de flujo	Java
(i,i[1-n])		int i=1;

(Suma,d[1-n])	Suma = 0	decimal Suma=0;

Los arreglos se definen en el lenguaje Java utilizando new. La sintaxis es la siguiente.

tipo[] *ariable ne tipo Elementos* ;

Donde *ariable* es un nombre de variable válido, *tipo* es un tipo de datos reconocido por el lenguaje, y *Elementos* es el número de elementos del arreglo. En Java los arreglos son de base cero, es decir, que los subíndices son 0, 1, 2, 3, y así. Declarar un arreglo como Numeros(3), genera un arreglo de 3 elementos con los subíndices 0,1,2 .

A continuación unos ejemplos.

```
int[ ] Ventas = new int[3];
int[ ] Ventas = new int[ ] {20000,15000};
```

Operadores principales

Los operadores principales del lenguaje Java, aplicables a los ejercicios tratados en el libro, son los siguientes.

Tipo	Operador	Nombre
Asignación	=	Igual
Aritmético	+	Suma/Concatenación de cadenas
	-	Resta
	*	Multiplicación
	/	División
	^	Exponenciación
Lógico		Disyuntiva
	||	Conjuntiva
	!	Negación

Comparativos	>	Mayor que
	<	Menor que
	>=	Mayor o igual que
	<=	Menor o igual que
	==	Igual a
	<>	Distinto a
Asignación incluyente	+=	Suma incluyente
	-=	Resta incluyente

Cambios de estado

Los cambios de estado en el lenguaje Java se realizan utilizando el operador de asignación (=).

ariable = E presión

Donde *ariable* es el nombre de una variable previamente declarada y *E presión* es el valor que se le asigna a la variable. Estos son unos ejemplos.

```
AreaCirculo = Lado^2;
i = i + 1;
i += 1;
```

En el ejemplo, las últimas dos líneas hacen lo mismo.

Equivalencias

Miniespecificación	Diagrama de flujo	Java
Promedio ← Suma/5	Promedio = Suma / 5	Promedio = Suma/5;
Suma ← Suma + Entrada(i)	Suma = Suma + Entrada(i)	Suma += Entrada(i);
i ← i + 1	i = i + 1	i += 1;

Mostrado de datos

El mostrado de datos se realiza en el lenguaje de Java a través de las órdenes System.out.print y System.out.println. La terminación "ln" de los comandos aplica si deseamos que al final de la instrucción se incluya un salto de línea.

```
System.out.print("E presión ");
```

Donde *E presión* es la expresión a mostrar.

Ejemplo.

```
System.out.print("Dame un número: ");
```

Equivalencias

Miniespecificación
« "Suma: " + Suma
Diagrama de flujo
Java
`System.out.println("Suma:"+Suma);`

Peticiones de datos

Para poder manejar una entrada de datos desde la consola, Java requiere de establecer un *flujo de datos* entre la interfaz y el programa; a este flujo se le da el nombre de *stream*. Además del flujo de datos requiere de un espacio temporal de memoria de trabajo, llamado *buffer*, para almacenar los datos que están siendo ingresados al flujo de datos. Para manejar esos supuestos es necesario instanciar un *lector de datos bufferizado* (BufferReader), asignándole lo que un flujo de datos reciba a través del sistema (InputStreamReader(System.In)). Antes de poder realizar peticiones de datos es necesario declarar un *manejador de entrada de datos*, colocando lo siguiente.

```
BufferedReader  ariable = new BufferedReader(new InputStreamReader(System.in));
```

Donde *ariable* es el nombre del manejador de entrada de datos que utilizaremos en cada operación de petición de datos al usuario. Esta línea requiere que se haya importado java.io.

Las peticiones de datos se realizan en el lenguaje de Java a través de la orden readLine. La sintaxis es la siguiente.

```
 ariable=  anejador.readLine();
```

Donde *ariable* es la variable en la que se almacenará lo que el usuario introduzca a la interfaz. *anejador* es el nombre del manejador de entrada de datos previamente definido, que se encargará de recibir los datos desde la interfaz. Vea cómo readLine se subordina completamente al manejador de entrada de datos.

Es muy importante hacer notar que readLine adquiere datos de tipo String. En caso de que se estén preguntando números, es necesario realizar la conversión al asignar valores a una variable numérica.

Si se pregunta un dato tipo...	...Se debe preguntar de la siguiente forma
byte	Byte.parseByte(*tring*)
long	Long.parseLong(*tring*)
int	Integer.parseInt(*tring*)

Donde *tring* es la expresión que será convertida al tipo numérico correspondiente.

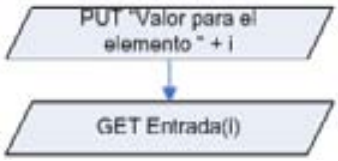
Ejemplos (se asume un manejador de datos de entrada, llamado entrada).

```
Mensaje=entrada.readLine();
Numero = Integer.parseInt(entrada.readLine());
```

Generalmente las peticiones de datos se realizan a solicitud, por lo que es común asociar una petición con un mostrado de datos previo, quedando como sigue.

```
System.out.print("E  presión");
 ariable = entrada.readLine();
```

Equivalencias

Miniespecificación
« "Valor para el elemento " + i » Entrada(i)
Diagrama de flujo

Java
`System.out.println("Valor para el elemento " + i);` `Entrada(i) = Integer.parseInt(entrada.readLine());`

Estructuras de decisión

Las estructuras de decisión se realizan en el lenguaje Java a través de la estructura if.

```
If   ondición
{
        nstrucciones por verdadero
[    } else {
        nstrucciones por falso ]
}
```

Donde *ondición* es la expresión lógica a evaluar. En caso de que *ondición* sea True, se ejecutará el bloque de código que contiene las *nstrucciones por verdadero*. De manera opcional se puede especificar un bloque de código a ejecutar si *ondición* resuelve por falso, utilizando else. En ese caso, se ejecutarían las *nstrucciones por falso*. La estructura if siempre debe terminar.

Ejemplos.

```
if (Monto > LimiteCredito)
{
        Venta = “Rechazada”;
}
else
{
        Venta = “Aprobada”;
}
```

Equivalencias

Miniespecificación

```
Si Entrada(i) >= Promedio Entonces
        « “Numero ” + Entrada(i) + “ mayor o igual al promedio”
De lo contrario
        « “Numero ” + Entrada(i) + “ menor al promedio”
Fin Si
```

Diagrama de flujo

Java

```
if  (Entrada(i) >= Promedio)
{
        System.out.println(“Numero ” + Entrada(i) +
                “ mayor o igual al promedio”);
}
else
{
        System.out.println(“Numero ” + Entrada(i) +
                “ menor al promedio”);
}
```

Estructuras de control

Las estructuras de control se realizan en el lenguaje Java a través de la estructura do while.

Es importante señalar la forma en que se manejan las estructuras de control con condición al final en Java, dado que las miniespecificaciones que pretendemos codificar están diseñadas de esa forma.

Para empezar, una estructura de control con condición al final representa una estructura de control de tipo until (hasta que). Dado que Java no posee estructura until (hasta que), se deberá especificar una condición que mantenga el proceso ejecutándose, mientras la condición se esté cumpliendo.

Supongamos que tenemos un bucle del 1 al 10. En las estructuras until la condición provocaba la terminación del ciclo, de ahí que el proceso se mantenía ejecutando hasta que 1 > 10. Con while la condición tendría que cambiarse a 1 <=10, es decir, mientras sea menor o igual a 10.

```
do
{
        nstrucciones

while (  ondición);
```

Donde *nstrucciones* es el bloque de código que se estará repitiendo hasta que *ondición* se cumpla. Todo lo que esté entre do y while se considera código controlado. Es importante no olvidar colocar y cambiar el estado de la variable de trabajo, dentro del código de control.

Ejemplos.

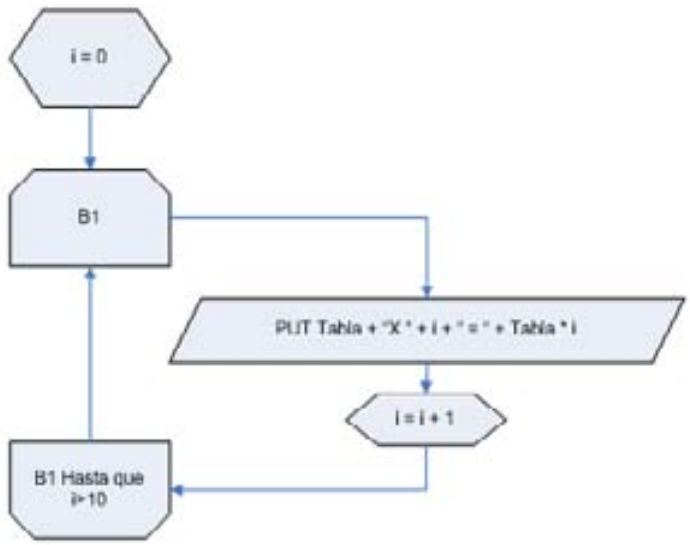
```
i = 1;
do
{
        System.out.println("Valor del contador: " + i);
        i += 1;
}
while (i <= 5);
```

Equivalencias

Miniespecificación

```
i ← 1
Inicio
        « Tabla + "X" + i + "=" + Tabla*i
        i ← i + 1
Hasta que i > 10
```

Diagrama de flujo



Java

```
i = 1;
do
{
        System.out.println(Tabla + "X" + i + "=" + Tabla*i);
        i += 1;
}
while (i <= 10);
```

Vea cómo la condición del bucle cambia a la condición opuesta, dado que cambiamos de un enfoque until a un enfoque while.

Aunque no manejamos estructuras de control con condición al inicio en este libro, proporcionamos la sintaxis de su implementación en Java.

```
while (  ondición)
{
        nstrucciones
```

Donde *nstrucciones* es el bloque de código que se estará repitiendo mientras que *ondición* se cumpla por verdadero.

Compilación y ejecución de programas

La compilación de los programas se realiza utilizando el compilador de línea de comandos de Java, llamado javac.exe. La sintaxis básica de compilación es la siguiente.

```
javac Programa.java
```

Donde *Programa* es el programa a compilar.

El comando se ejecuta desde la línea de comandos. Es importante no omitir la extensión de los programas desarrollados en Java (.java), dado que la compilación generaría error.

Al concluir la compilación sin errores, se habrá creado un programa de extensión .class, que se trata de *código intermedio de ava*, llamado también *bytecode*; sólo se podrá ejecutar el ensamblado en equipos con JVM instalado.

El programa podrá ejecutarse desde la línea de comandos, invocando el intérprete de programas Java. Su sintaxis es la siguiente.

```
java Programa
```

Donde *Programa* es el nombre del archivo .class que desea ejecutar. Al ejecutar el código intermedio, JVM se encargará de generar el código máquina optimizado para el equipo desde el cual se intentó la ejecución, mostrando finalmente los resultados del programa. Es importante que no incluya la extensión .class al intérprete de comandos.

www.detodoprogramacion.com

Ejercicio 13.01

Codificación de C1Mensaje en Java

En el presente ejercicio se codificará el primer caso de estudio, utilizando el lenguaje Java.

1. Utilice un editor de texto plano (*loc de notas*, por ejemplo), para escribir el código fuente llamado C1Mensaje.java; el archivo deberá guardarse en el directorio de trabajo del libro (C:\APIP).

Este programa tiene como finalidad comprobar técnicas y herramientas de desarrollo. Lo que el programa debe hacer es preguntar por un mensaje de saludo, para posteriormente mostrar el mensaje que se le proporcionó.

Miniespecificación

```
1   INICIO:
2           (Mensaje, x(30))
3           « "Escribe un mensaje a mostrar: "
4           » Mensaje
5           « Mensaje
6   :FIN
```

Codificación de **C1Mensaje.java**

```
1   import java.io.*;
2   import java.lang.*;
3
4   class C1Mensaje
5   {
6           public static void main(String args[])
7                   throws IOException
8       {
9
10                  BufferedReader entrada = new
11                          BufferedReader(new InputStreamReader(System.in));
12
13                  String Mensaje;
14                  System.out.print("Escribe un mensaje a mostrar: ");
15                  Mensaje=entrada.readLine();
16                  System.out.println(Mensaje);
17
18          }
19  }
```

2. Compile el programa utilizando la siguiente línea.

```
javac C1Mensaje.java
```

3. Si no genera errores, ejecute el programa de la siguiente forma.

```
java C1Mensaje
```

Si todo ocurrió como se esperaba, la salida será la siguiente:

Entrada	Escribe un mensaje a mostrar: Aprenda
Salida	Aprenda

FIN DEL EJERCICIO ✱

Ejercicio 13.02

Codificación de C2Niveles en Java

En el presente ejercicio se codificará el segundo caso de estudio, utilizando el lenguaje Java.

1. Utilice un editor de texto plano (*loc de notas*, por ejemplo), para escribir el código fuente llamado C2Niveles.java; el archivo deberá guardarse en el directorio de trabajo del libro (C:\APIP).

Se desea saber la disponibilidad del servicio de Internet para una persona, expresado en horas de disponibilidad.

Dicho nivel de servicio depende del grado de disponibilidad que ofrece la computadora que utiliza, así como del grado de disponibilidad del enlace a Internet que utiliza la máquina.

El horario de labores de la persona que utiliza la computadora y el enlace a Internet es de 8 horas.

Para poder realizar los cálculos adecuadamente, tome en cuenta que la disponibilidad de un sistema de dispositivos (dos o más dispositivos que trabajan de forma conjunta) se calcula multiplicando los porcentajes de

www.detodoprogramacion.com

grado de disponibilidad de cada uno de ellos, y el resultado siempre será menor a la disponibilidad del menor.

Para calcular el tiempo de disponibilidad de un servicio (en este caso el acceso a la información publicada en Internet), sólo basta con multiplicar el tiempo en que es requerido el servicio por la disponibilidad del sistema de dispositivos.

Miniespecificación

```
1   INICIO:
2           (DC,d[0-1])
3           (DE,d[0-1])
4           (TL,d[0-1])
5           (DS,d[0-1])
6           (DH,d[0-1])
7           « "Grado de disponibilidad de la computadora:"
8           » DC
9           « "Grado de disponibilidad del enlace:"
10          » DE
11          « "Tiempo laborable en horas:"
12          » TL
13          DS ← DC * DE
14          DH ← DS * TL
15          « "La disponibilidad del servicio de Internet es de " + DH + " horas"
16  :FIN
```

Codificación de **C2Niveles.java**

```
1   import java.io.*;
2   import java.lang.*;
3
4   class C2Niveles {
5           public static void main(String args[])
6           throws IOException
7       {
8
9                   BufferedReader entrada = new
10                          BufferedReader(new InputStreamReader(System.in));
11
12                  float DC;
13                  float DE;
14                  float TL;
15                  float DS;
16                  float DH;
17
18                  System.out.print("Grado de disponibilidad de la computadora: ");
19                  DC=Float.parseFloat(entrada.readLine());
20                  System.out.print("Grado de disponibilidad del enlace: ");
```

```
21                  DE=Float.parseFloat(entrada.readLine());
22                  System.out.print("Tiempo laborable en horas:  ");
23                  TL=Float.parseFloat(entrada.readLine());
24
25                  DS= DC*DE;
26                  DH= DS*TL;
27
28                  System.out.println("La disponibilidad del servicio de internet es de "+
29                          DH +" horas");
30
31          }
32  }
```

2. Es importante analizar que al preguntar los datos compile el programa utilizando la siguiente línea.

```
javac C2Niveles.java
```

3. Si no genera errores, ejecute el programa de la siguiente forma.

```
java C2Niveles
```

Si todo ocurrió como se esperaba, la salida será la siguiente:

Entrada	Grado de disponibilidad de la computadora: .98
Entrada	Grado de disponibilidad del enlace: .85
Entrada	Tiempo laborable en horas: 8
Salida	La disponibilidad del servicio de internet es de 6.6640 horas

FIN DEL EJERCICIO ✱

Ejercicio 13.03

Codificación de C3Tabla en Java

En el presente ejercicio se codificará el tercer caso de estudio, utilizando el lenguaje Java.

1. Utilice un editor de texto plano (*loc de notas*, por ejemplo), para escribir el código fuente llamado C3Tabla.java; el archivo deberá guardarse en el directorio de trabajo del libro (C:\APIP).

El programa debe mostrar cualquier tabla de multiplicar que le solicitemos. La única restricción es que la tabla deberá ser de un número mayor o igual a 1, y menor o igual a 10.

Si se pide una tabla de un número que no esté entre el 1 y el 10, el programa debe indicar, mediante un mensaje, que no puede continuar.

Miniespecificación

```
1   INICIO:
2           (i,i[0-n])
3           (Tabla,i[1-10])
4           « "Qué tabla deseas: "
5           » Tabla
6           Si Tabla >= 1 ∧ Tabla <=10 Entonces
7                   i ← 1
8                   Inicio
9                           « Tabla + " X" + i + "=" + Tabla * i
10                          i ← i + 1
11                  Hasta que i > 10
12          De lo contrario
13                  « "Válido sólo del 1 al 10"
14          Fin Si
15  :FIN
```

Codificación de **C3Tabla.java**

```
1   import java.io.*;
2   import java.lang.*;
3
4   class C3Tabla
5   {
6           public static void main(String args[])
7           throws java.io.IOException
8     {
9
10                  BufferedReader entrada = new
11                          BufferedReader(new InputStreamReader(System.in));
12                  int i=0;
13                  int Tabla;
14
15                  System.out.print("Qué tabla deseas: ");
16                  Tabla=Integer.parseInt(entrada.readLine());
17
```

```
18                  if (Tabla >=1 & Tabla<=10 )
19                  {
20
                            do
                            {
21
22                                  System.out.println(Tabla + " X " + i + " = " + Tabla*i );
23                                  i+=1;
24                          }
25                          while (i <= 10);
26                  }
27                  else
28                  {
29                          System.out.println("Valido solo del 1 al 10");
30                  }
32
33          }
34  }
35
36
```

2. Compile el programa utilizando la siguiente línea.

```
javac C3Tabla.java
```

3. Si no genera errores, ejecute el programa de la siguiente forma.

```
java C3Tabla
```

Si todo ocurrió como se esperaba, la salida será la siguiente:

Entrada	Que tabla deseas: 5
Salida	5 X 1 = 5
	5 X 2 = 10
	5 X 3 = 15
	5 X 4 = 20
	5 X 5 = 25
	5 X 6 = 30
	5 X 7 = 35
	5 X 8 = 40

	5 X 9 = 45 5 X 10 = 50

Entrada	Que tabla deseas: 25
Salida	Válido sólo del 1 al 10

FIN DEL EJERCICIO ✱

Ejercicio 13.04

Codificación de C4MultiTabla en Java

En el presente ejercicio se codificará el cuarto caso de estudio, utilizando el lenguaje Java.

1. Utilice un editor de texto plano (*loc de notas*, por ejemplo), para escribir el código fuente llamado C4MultiTabla.java; el archivo deberá guardarse en el directorio de trabajo del libro (C:\APIP).

El programa debe mostrar todas las tablas de multiplicar del 1 al 5, incluyendo un encabezado por tabla.

Miniespecificación

```
1   INICIO:
2           (i,i[0-n])
3           (y,i[0-n])
4           i ← 1
5           Inicio
6                   « "TABLA: " + i
7                   y ← 1
8                   Inicio
9                           « i + " X" + y + "=" + i * y
10                          y ← y + 1
11                  Hasta que y > 10
12                  i ← i + 1
13          Hasta que i > 5
14  :FIN
```

Codificación de **C4MultiTabla.java**

```
1   import java.io.*;
2   import java.lang.*;
3
4   class C4MultiTabla
5   {
6           public static void main(String args[])
7           throws java.io.IOException
8     {
9
10                  int i=1;
11                  int y=1;
12
13                  i=1;
14                  do
15                  {
16                          System.out.println("Tabla: " + i );
17                          y=1;
18                          do
19                          {
20                                  System.out.println(i + " X " + y + " = " + i*y );
21                                  y += 1;
22                           }
23                          while (y <= 10);
24                          i += 1;
25                  }
26                  while (i <= 5);
27
28    }
29  }
```

2. Compile el programa utilizando la siguiente línea.

```
javac C4MultiTabla.java
```

3. Si no genera errores, ejecute el programa de la siguiente forma.

```
java C4MultiTabla
```

Si todo ocurrió como se esperaba, la salida será la siguiente:

Salida	Tabla: 1 1 X 1 = 1

```
1 X 2 = 2
1 X 3 = 3
1 X 4 = 4
1 X 5 = 5
1 X 6 = 6
1 X 7 = 7
1 X 8 = 8
1 X 9 = 9
1 X 10 = 10
Tabla: 2
2 X 1 = 2
2 X 2 = 4
2 X 3 = 6
2 X 4 = 8
...

5 X 7 = 35
5 X 8 = 40
5 X 9 = 45
5 X 10 = 50
```

FIN DEL EJERCICIO ✱

Ejercicio 13.05

Codificación de C5Promedios en Java

En el presente ejercicio se codificará el quinto caso de estudio, utilizando el lenguaje Java.

1. Utilice un editor de texto plano (*loc de notas*, por ejemplo), para escribir el código fuente llamado C5Promedios.java; el archivo deberá guardarse en el directorio de trabajo del libro (C:\APIP).

El programa debe pedir 5 números, para después mostrarlos indicando si están o no por encima de su promedio. Al final deberá enviarse un resumen en donde se indique la suma total de los números, y el promedio calculado.

Miniespecificación

```
1   INICIO:
2           (i,i[0-n])
3           (Entrada(5),i[0-n])
4           (Suma,d[0-n])
5           (Promedio,d[0-n])
6           Suma ← 0
7           i ← 1
8           Inicio
9                   « "Valor para el elemento " + i + ": "
10                  » Entrada(i)
11                  Suma ← Suma + Entrada(i)
12                  i ← i + 1
13          Hasta que i > 5
14          Promedio ← Suma / 5
15          i ← 1
16          Inicio
17                  Si Entrada(i) >= Promedio Entonces
18                      « "Número " + Entrada(i) + " mayor o igual al promedio: "
19                  De lo contrario
20                      « "Número " + Entrada(i) + " menor al promedio: "
21                  Fin Si
22                  i ← i + 1
23          Hasta que i > 5
24          « "Suma: " + Suma
25          « "Promedio: " + Promedio
26  :FIN
```

Codificación de **C5Promedios.java**

```
1   import java.io.*;
2   import java.lang.*;
3
4   class C5Promedios
5   {
6           public static void main(String args[])
7           throws IOException
8       {
9
10                  int i;
11                  int[] Entrada = new int[5];
12                  int Suma;
13                  float Promedio;
```

```
14                  BufferedReader entrada = new
15                          BufferedReader(new InputStreamReader(System.in));
16
17                  Suma = 0;
18
19                  i=1;
20                  do
21                  {
22                          System.out.print("Valor para el elmento "+  i + ": ") ;
23                          Entrada[i-1] = Integer.parseInt(entrada.readLine());
24                          Suma += Entrada[i-1];
25                          i += 1;
26                  }
27                  while (i <= 5);
28
29                  Promedio=Suma/5;
30
31                  i=1;
32                  do
33                  {
34                          if (Entrada[i-1] >= Promedio)
35                          {
36                                  System.out.println("Numero " + Entrada[i-1] +
37                                          " mayor igual al promedio.");
38                          }
39                          else
40                          {
41                                  System.out.println("Numero " + Entrada[i-1] +
42                                          " menor al promedio.");
43                          }
44                          i += 1;
45                  }
46                  while (i <= 5);
47
48                  System.out.println("Suma: " + Suma );
49                  System.out.println("Promedio: " + Promedio );
50
51          }
52  }
```

2. Es importante mencionar que este programa tiene discrepancias con la miniespecificación, provocadas por el cambio de enfoque en las estructuras de control. El hecho de cambiar las estructuras de tipo until, por estructuras de tipo while, hace que las condiciones que eran i > 5 en la miniespecificación (líneas 13 y 23) pasen a ser i<=5 en el programa (líneas 27 y 46).

3. En las llamadas al arreglo, en la miniespecificación se coloca como subíndice i (líneas 11, 17, 18 y 20), mientras que en el programa se refiere al subíndice como i-1 (líneas 23, 24, 34, 36, 41). Esto se origina porque para la miniespecificación el primer elemento del arreglo tiene el subíndice 1, pero

para el programa el primer elemento tiene el subíndice 0; para la miniespecificación el segundo elemento tiene el subíndice 2, y para el programa es 1. Si se observa, el subíndice del programa es igual al subíndice de la miniespecificación, menos 1. Al colocar en el programa el subíndice como i-1 homologamos el funcionamiento de la miniespecificación y del programa, y de esa forma el resto de la miniespecificación sigue funcionando sin cambio, además que el usuario no se percata del cambio.

4. Compile el programa utilizando la siguiente línea.

```
javac C5Promedios.java
```

5. Si no genera errores, ejecute el programa de la siguiente forma.

```
java C5Promedios
```

Si todo ocurrió como se esperaba, la salida será la siguiente:

Salida

```
Valor para el elemento 1: 1
Valor para el elemento 2: 2
Valor para el elemento 3: 3
Valor para el elemento 4: 4
Valor para el elemento 5: 5
Número 1 es menor al promedio
Número 2 es menor al promedio
Número 3 es mayor o igual al promedio
Número 4 es mayor o igual al promedio
Número 5 es mayor o igual al promedio
Suma: 15
Promedio: 3
```

FIN DEL EJERCICIO ✱

Ejercicio 13.06

Desarrollando un programa de regla de tres simple usando Java

Utilizando Java, realice las siguientes actividades:

- ❒ Desarrolle el código fuente.
- ❒ Asigne como nombre del programa TresSimple.java .
- ❒ Compílelo.
- ❒ Ejecútelo.

Elabore un programa que realice el cálculo de una regla de tres simple. Como sabe, dicha regla permite calcular la proporción de un número respecto a otro que actúa como referencia de la proporción. Este programa contesta preguntas del tipo ¿qué porcentaje de 200 es 135? El 200 sería el número base de la proporción, mientras que el 135 sería la proporción. La solución se calcula multiplicando la proporción por 100, y dividiéndolo entre el número base de la proporción.

El programa deberá mostrar el resultado del cálculo.

FIN DEL EJERCICIO ✱

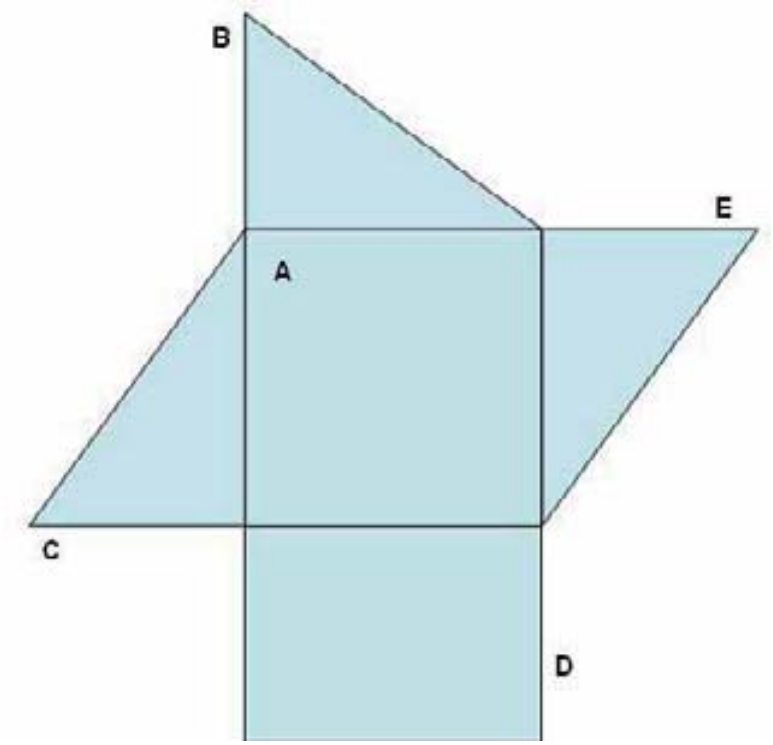
Ejercicio 13.07

Desarrollando un programa para el cálculo del área de un polígono irregular usando Java

Utilizando Java, realice las siguientes actividades:

- ❒ Desarrolle el código fuente.
- ❒ Asigne como nombre del programa Poligono.java .
- ❒ Compílelo.
- ❒ Ejecútelo.

Observe la siguiente figura.



El programa debe calcular el área del polígono. La figura se compone de un cuadrado perfecto (A), tres triángulos rectángulos iguales (B, C, E), cuya base mide lo mismo que uno de los lados del cuadrado, y un rectángulo cuyo ancho mide lo mismo que uno de los lados del cuadrado, mientras que la altura mide lo mismo que la altura de los triángulos. Se pide que haga un programa que calcule el área total.

Por cultura general, sabemos que el área de un rectángulo se calcula multiplicando su base por su altura; el área de un triángulo rectángulo se calcula multiplicando su base por su altura, y luego dividiendo el resultado entre dos; finalmente, el área de un cuadrado se calcula elevando al cuadrado la medida de uno de sus lados.

FIN DEL EJERCICIO ✱

Ejercicio 13.08

Desarrollando un programa con bucles, condicionales y arreglos usando Java

Utilizando Java, realice las siguientes actividades:

- ❒ Desarrolle el código fuente.
- ❒ Asigne como nombre del programa Descuento.java .
- ❒ Compílelo.
- ❒ Ejecútelo.

Una tienda departamental ofrece descuentos directos sobre las compras que los clientes realizan.

El descuento normal va en función al tipo de cliente que la persona sea: A los clientes normales se les ofrece un 10 de descuento, y a los clientes premier un 15 de descuento.

Actualmente se tiene una promoción: en la compra de 5 artículos, se ofrecerá un descuento adicional al normal. Si el monto de la compra no excede de 1,000 pesos, se agrega un 5 de descuento; si el monto de la compra está entre 1,000 y 2,000 pesos, se agrega un descuento del 10 ; si la compra es mayor a 2,000 pesos, entonces el descuento adicional será del 15 .

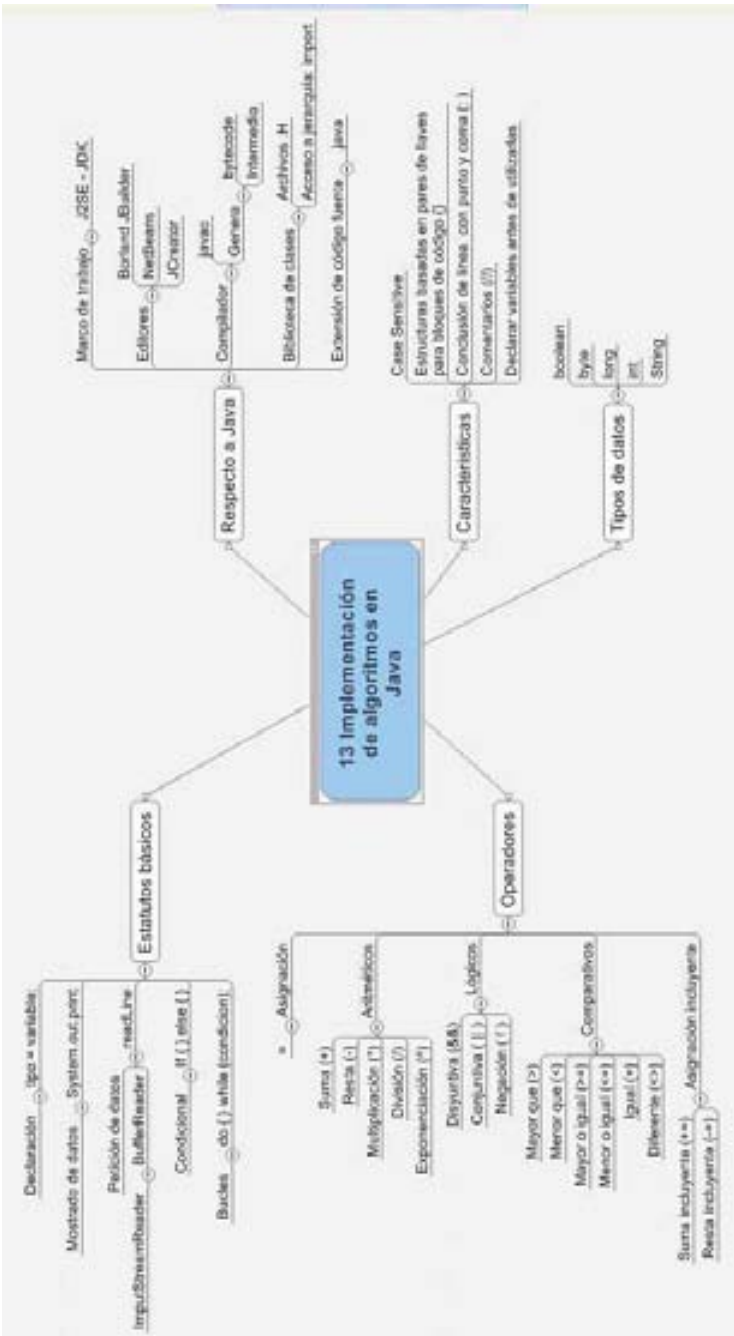
En ese sentido, el menor descuento que se puede obtener es cuando un cliente normal compra menos de 1,000 pesos (15), y el máximo descuento que se puede obtener es cuando un cliente premier compra más de 2,000 pesos (30).

El programa deberá pedir la información de los precios de 5 productos, y determinar el monto de descuento aplicable.

Utilice arreglos.

FIN DEL EJERCICIO ✱

Mapa mental del capítulo



Terminología

.class, 369
Borland Jbuilder, 369
Buffer, 375
BufferReader, 375
Bytecode, 369
Código intermedio de Java, 381
Importar paquetes, 370
J2SE, 368
Java 2 Plataform Standard Edition, 368
Java Development Kit, 368
Java Foundation Class, 369
Java Virtual Machine, 369
JAVA.EXE, 369
JAVAC.EXE, 369
JCreator, 369
JDK, 368
JFC, 369
JVM, 369
Lector de datos bufferizado, 375
Manejador de entrada de datos, 375
NetBeans, 369
Packages, 369
Paquetes, 369

Preguntas

13.1 Tome tiempo de los siguientes aspectos:

13.1.1 ¿Tiempo en que se desarrollaron los casos 6, 7 y 8 en Java?

13.1.2 ¿Veces en que fue necesario corregir cada uno de los programas 6, 7 y 8, a fin de corregir el programa por errores de sintaxis?

13.1.3 ¿Qué tan fácil fue desarrollar los casos 6, 7 y 8 en Java, en su opinión?

13.1.4 ¿Qué tan útiles fueron las miniespecificaciones que desarrolló, para efectos de construir su código en Java? ¿Requirieron ajustes?

13.2 a ha visto y probado los más poderosos lenguajes de la actualidad. Compare la experiencia que tuvo con cada uno de los lenguajes, y decida qué lenguaje es el que más le convence. Su decisión no descansará en comentarios o recomendaciones: Usted ya probó la tecnología. ¿Con cuál se siente más a gusto?

13.3 Dadas las diferencias que usted percibió entre uno y otro lenguaje ¿Considera justificadas las diferencias que existen entre los programadores de Java y Visual Basic.NET? ¿O ha comprobado que el lenguaje es irrelevante?

Examen rápido

Parte 1 : Marque co n una cruz el inciso que considere que responde a la pregunta.

1) Es el marco de trabajo necesario para trabajar con programas desarrollados en Java
(a) .NET Frame or
(b) J2SE
(c) No se requiere un marco de trabajo

2) El compilador de Java es
(a) Bcc32
(b) Vbc.exe
(c) Javac

3) La biblioteca de clases que utiliza Java se llama
(a) JFC
(b) BCL
(c) No tiene nombre

4) Al compilar un programa en Java se genera un archivo de extensión
(a) E E
(b) Class
(c) OBJ

5) Al compilar un programa en Java el archivo que se genera es llamado
(a) assembly
(b) bytecode
(c) machine code

Parte 2: Coloque en la línea la letra «F» si la sentencia es falsa, y «V» si la sentencia es verdadera.

_____ **6)** El compilador de Java es en dos fases.

_____ **7)** Java soporta bucles de tipo Until.

_____ **8)** NetBeans es una herramienta de desarrollo asociada a Java.

_____ **9)** La biblioteca de clase de Java se compone de archivos de extensión .H

_____ **10)** En Java para tener acceso a la biblioteca de clases es necesario referir el nombre físico de la biblioteca, dentro del programa, usando Imports.

1

Fundamentos de la programación orientada a objetos

Objetivos: Aprenderá los conceptos fundamentales de la programación orientada a objetos, así como las ventajas de dicha orientación. Aprenderá su implementación en Visual Basic.NET

1. Aprenderá los términos fundamentales de la p rogramación orientada a objetos: clases, objetos, campos, propiedades, métodos y eventos.
2. Aprenderá en qué consiste el enca psulamiento, la h erencia y el polimorfismo.
3. Aprenderá a implementar clases, y sus capacidades de encapsulamiento, herencia y polimorfismo, usando Visual Basic.NET.

www.detodoprogramacion.com

Contenido

Fundamentos de la programación orientada a objetos 401
Términos básicos de POO 403
Clases y objetos 403
Propiedades, métodos y eventos 403
Encapsulamiento (encapsulation) 404
Herencia (inheritance) 405
Polimorfismo (polymorphism) 406
Overloading, Overriding, y Shadowing 406
Elementos esenciales de los objetos que permiten su programación 407
Identidad 407
Estado 408
Comportamiento 409
Definición de una clase 410
Definición de propiedades 411
Creando una clase y generando una instancia 413
Definiendo y utilizando propiedades 414
Herencia 417
¿Cuándo se debe utilizar la herencia? 417
Inherits (Heredar) 418
NotInheritable (No Heredable) 418
MustInherit (Debe Heredar) 418
Modificadores de acceso, necesarios para la herencia 419
Estatutos auxiliares de herencia 419
NotOverridable (No Remplazable) 420
Overridable (Remplazable) 420
MustOverride (Debe Remplazarse) 420
Overrides (Remplaza) 421
Overloads (Sobrecarga/Sustituye) 421
MyBase y MyClass 421
Polimorfismo 422
Creando una jerarquía de clases por herencia 422
Comprobando el polimorfismo basado en herencia, y el uso de MyBase 424
Mapa mental del capítulo 429
Terminología 430
Preguntas 430
Examen rápido del módulo 431

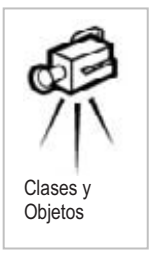
En el presente capítulo se tratan los conceptos generales para la *programación orientada a objetos* (*POO*), que están relacionados con los objetos en sí, y con su relación con otros objetos.

Todos los lenguajes que trata este libro son orientados a objetos. Java y C++ siempre fueron orientados a objetos. Visual Basic en la versión 6 no soportaba la orientación a objetos, y ya en la versión .NET, finalmente soportó los rasgos característicos de la programación a objetos, que son la herencia, el polimorfismo y el encapsulamiento. C# nació siendo orientado a objetos, como todos los lenguajes .NET.

Decidimos comprobar la programación orientada a objetos usando Visual Basic.NET, por seleccionar alguno de los lenguajes. Demostrarlo en todos los lenguajes sería muy extenso y complicado.

Si se pregunta qué es mejor, si el enfoque de procedimientos, el de eventos o el orientado a objetos, la respuesta es contundente: depende de lo que quiera hacer. Hay cosas más sencillas de implementar en una orientación que otra; dicho esto, no se sienta mal por no haber aprendido a programar en POO con anterioridad. Quizá no lo haya necesitado.

Términos básicos de POO


Clases y Objetos

Clases y objetos

Una *lase* es una definición formal de un tipo de objeto. La clase define qué datos formarán parte de un objeto, qué tareas desarrollará el objeto, y de qué manera interactuará el objeto con el usuario y con otros objetos.

Los *ob etos* son instancias de una clase. Por *instan ia* podemos entender una copia funcional de la clase. A la acción de generar un objeto a partir de una clase, se le denomina *instan ia ión*.

Al procedimiento o código que realiza la instanciación se le denomina *onstru tor*. Al procedimiento o código que destruye a un objeto, liberando los recursos que éste consumía, se le llama *destru tor*.

Propiedades, métodos y eventos

Las clases, y por tanto los objetos que se derivan de ellas, son un conjunto de datos y comportamientos; tanto las clases como los objetos están formados por propiedades y métodos.

Las *propiedades* (*properties*) son características asociadas a un objeto, expresadas en valores, que determinan su apariencia y posibilidades de funcionamiento.

Los *todos* son los comportamientos predefinidos que puede presentar un objeto. En cierta forma, los métodos representan las acciones que el objeto podrá realizar. La manera en que se pueden definir los métodos es agregando procedimientos y funciones a una clase; al momento de generar una instancia de dicha clase, los procedimientos y funciones que le codificamos a la clase formarán los métodos disponibles para el objeto.

Un *e ento* es el resultado de la interacción que un usuario o programa tiene con un objeto, en una circunstancia dada, que desencadena la ejecución de un procedimiento o método.

A las propiedades, métodos y eventos también suele llamárseles *miembros* (*members*), que son el conjunto de elementos declarados por una clase.

Para que una aplicación realmente esté orientada a objetos, es necesario que soporte las cualidades de encapsulamiento, herencia y polimorfismo. Estas tres cualidades son representativas de POO.

Encapsulamiento (encapsulation)

El *en apsula iento* es la capacidad que tiene un objeto para almacenar datos y procedimientos, como una *unidad un ional*, y permitir el acceso a ellos sin que los demás objetos tengan la necesidad de conocer cómo se integran dichos datos y dicho comportamiento dentro del objeto. El diálogo entre los objetos se da a través de paquetes de datos, aceptan peticiones de acción o información, y actúan o retornan la información solicitada.

El encapsulamiento sólo debe respetar una regla: al modificar un objeto éste debe seguir aceptando los mismos mensajes de petición, y enviando los mismos mensajes de respuesta, ya que de otra forma se requiere modificación en la programación, que siempre es laboriosa. Fuera de eso, el objeto puede ser una caja negra para los demás desarrolladores, sin que ello perjudique la funcionalidad de la aplicación.

Las ventajas que proporciona el encapsulamiento a los programadores de soft are son las siguientes:

- **Facilita la modularidad.** Un *ódulo* es un bloque de código que desarrolla un proceso específico. La *odularidad* es la capacidad que se tiene de dividir los procesos en múltiples procesos más pequeños.
- **Mantiene la información oculta.** El objeto puede por si solo encargarse de modificar los valores de sus propiedades; además, para hacer uso del objeto no es necesario conocer cómo es que realiza las operaciones (comportamiento). Sólo el objeto se encarga de manejar su estado y su comportamiento.

Herencia (inheritance)

La *eren ia* describe la habilidad de crear una nueva clase basada en una clase existente.

La clase existente a partir de la cual se generará la nueva clase recibe el nombre de *lase base* (*base lass*); la nueva clase, llamada *lase deri ada* (*deri ed lass*), hereda todas las propiedades, métodos y eventos de la clase base, y a partir de ahí, puede agregársele las propiedades y métodos particulares que la hacen especial.

Un ejemplo puede aclararnos las cosas. Imagine que crea una clase llamada `VehículoAutoMotor`. Esta clase contendrá propiedades que todos los vehículos auto motores (automóviles) tienen, como son color, tipo de transmisión, tipo de combustible que utilizan, etcétera. Además, tendrá definidos los métodos aplicables para todos los vehículos auto motores, por ejemplo, encender el motor o apagar el motor.

Hay un tipo de vehículos auto motores que sirven para carga de materiales (vehículo de carga); en ellos, la capacidad de carga en toneladas, las dimensiones máximas de la carga, y la necesidad de refrigeración de la carga son importantes, pero, ¿son esos datos importantes para un vehículo que sólo es utilizado para transportar personas? La respuesta es que no.

Tenemos las siguientes alternativas de solución para manejar la información del ejemplo:

- Definir la clase `VehículoAutoMotor` con todas las características posibles para todos los vehículos auto motores posibles; eso sobrecargaría a la clase de muchas propiedades no aplicables para todos los vehículos, complicando el trabajo de afirmar o negar las características aplicables a un vehículo en particular. No hay herencia; todo está siendo manejado en un mismo objeto, tal y como sería en una programación procedural.
- Definir una clase independiente de `VehículoAutoMotor`, por ejemplo `VehículoAutoMotorCarga`, que contenga todas las propiedades y los métodos que requieren ser especificadas para un vehículo auto motor de carga; obviamente, muchas de las características de los vehículos auto motores de carga son comunes para todos los vehículos auto motores. Eso implicaría una doble codificación (*odi i a ión redundante*), que revela que, no obstante que ya teníamos codificados los pormenores de los vehículos auto motores, no reutilizamos dicho código, pues creamos uno enteramente nuevo. En esta alternativa tampoco hay herencia.
- Definir una clase independiente, por ejemplo `VehículoAutoMotorCarga`, que contenga sólo aquellas propiedades y métodos no definidos en `VehículoAutoMotor`. La nueva clase (`VehículoAutoMotorCarga`) se generaría a partir de una clase base (`VehículoAutoMotor`), aprovechando toda su codificación. Esta alternativa implica herencia, en donde se tiene una clase base y una clase derivada.

Las ventajas que proporciona la herencia los programadores de soft are son las siguientes:

- **Reuso de código.** La mayor ventaja de la herencia es que el código se reutiliza de una forma más eficiente, pues las clases parten de otras clases más generales; esto garantiza que de inicio las clases ya tienen la funcionalidad de otras anteriormente creadas. Sólo se estará desarrollando nueva funcionalidad, evitándose el código redundante.
- **Jerarquía de clases.** Es posible realizar una jerarquía que distinga procesos generales y específicos.

A partir de una clase base surgen clases derivadas que pueden disponer de todas las propiedades, métodos y eventos de la clase base, y que además incluyen las características particulares de aquello que no es posible representar usando solamente lo definido en la clase base. En la jerarquía de clases, la clase base no requiere de las clases derivadas, pero las clases derivadas si requieren de lo definido en la clase base.

Polimorfismo (polymorphism)

El *poli or is o* es la capacidad de manejar múltiples clases que pueden ser utilizadas de manera intercambiable, a través de una misma implementación (nombre de clase). El polimorfismo es esencial para la programación orientada a objetos, ya que permite, a través de un mismo nombre de clase, poder agrupar diferente funcionalidad, dependiendo de las propiedades y métodos que se hagan uso.

Imagine que tiene una aplicación que controla las ventas en un negocio. Puede ser que tenga un objeto llamado `RegistraVenta`, que se encarga de almacenar en una base de datos los pormenores de la venta. Suponga que hay un objeto llamado `ActualizaLíneaCrédito`, que se encarga de actualizar la línea de crédito de los clientes que compran en el establecimiento; como ha de suponer, las acciones que realizará `ActualizaLíneaCrédito` tienen que ver si la compra se realiza a crédito o de contado. En enfoque procedural, sería necesario hacer dos procedimientos distintos, de diferente nombre, para controlar cada una de las alternativas.

En POO, se pueden generar dos clases distintas, que contengan cada una de las alternativas de la actualización de la línea de crédito; la diferencia estriba en que para el programa sólo existirá una. Será la misma plataforma de ejecución quien determine, con base a los datos proporcionados, qué clase es la que aplica.

Overloading, Overriding, y Shadowing

Para implementar el polimorfismo, hay que conocer tres conceptos básicos: Overloading, Overriding, y Shado ing. Al explicar el polimorfismo, concluimos que

un mismo nombre de método puede tener funcionalidad distinta dependiendo de los argumentos que le son proporcionados.

En Visual Basic.NET los argumentos van entre paréntesis y separados por comas, después del nombre del método o función; por tratarse de una lista de valores, estamos ante la presencia de una *ole ión*. Cada uno de los valores que forman una colección recibe el nombre de *ie bros* (*e ber*).

Los miembros con característica *erloaded* son utilizados para proporcionar diferentes versiones de una propiedad o método que tiene el mismo nombre, pero que acepta diferente número de argumentos, o argumentos de diferente tipo de dato.

Las propiedades y métodos con característica *erridden* son utilizados para remplazar una propiedad o método que, habiendo sido heredado de una clase base, no es lo apropiado para la clase derivada. Esto permite sustituir elementos heredados de la clase base, con aquellos que la clase derivada realmente necesita.

La única restricción que se tiene es que los miembros *erridden* deben aceptar el mismo número de argumentos y retornar los mismos valores, respetando los tipos de datos de la clase base, a efecto de que el encapsulamiento de los objetos sea posible.

Los miembros con característica *ado ed* son aquellos que remplazan localmente a otros miembros existentes, dentro de un alcance diferente al original.

Elementos esenciales de los objetos que permiten su programación

Al generar instancias de una clase, estamos generando un objeto. Una vez que los objetos existen, es posible que los utilicemos para obtener funcionalidad de la aplicación. Es importante que nos hagamos a la idea de que un objeto, una vez que es creado, ya posee vida propia y es independiente de todos los demás objetos.

Los objetos, una vez creados, tendrán tres características principales:

- Identidad.
- Estado.
- Comportamiento.

Identidad

La *identidad* del objeto es su capacidad de tener un nombre que lo identifica y lo diferencia de los demás objetos. No podemos crear ningún objeto sin asignarle un

nombre para su identificación, pues de ser así generaríamos un elemento al cual no podríamos invocar de ninguna manera.

Al momento de instanciarse, todos los objetos son exactamente iguales a la clase de la cual derivan; sólo se diferenciarán en su identidad.

En Visual Basic.NET los objetos poseen una propiedad llamada **Name**, a través de la cual se manifiesta la identidad de los objetos. A la propiedad **Name** también se le conoce como *propiedad de identidad*.

La propiedad **Name** puede ser utilizada en tiempo de ejecución para evaluar la identidad de un objeto por su nombre, y no por su identificación en el programa (nombre de la variable objeto, a lo que se conoce como *no bre pro ra ti o*). El valor que posee la propiedad **Name** es de tipo **String**, lo que facilita mucho su utilización en los procesos que estemos codificando.

Los programadores en versiones anteriores de Visual Basic tal vez no le dieran importancia al hecho de que todos los controles que se utilizaban en las aplicaciones debían ser referidos por su nombre programático, pues era la única posibilidad que había para identificar los elementos de un programa. Una vez realizada la codificación, cambiar la identidad a un control (modificar la propiedad **Name**) era impensable, ya que sería necesario modificar el código para soportar el nuevo nombre asignado.

En Visual Basic.NET queda claramente definido que se tiene un nombre programático (nombre de los objetos), y se tiene la identidad de los objetos, en la propiedad **Name**.

Estado

El *estado* de un objeto es la situación de forma y comportamiento que tiene el objeto en un momento determinado, en tiempo de ejecución.

El estado está determinado por el juego de valores que tienen sus propiedades; como sabemos, las propiedades son las características particulares del objeto, y especifican su nombre, su posición, su apariencia, y hasta sus capacidades de interacción con el usuario y con otros objetos.

Un objeto puede estar disponible o no disponible para su uso en una interfaz (`Enabled=True` o `Enabled=False`); puede tener un ancho de 10 pixels o de 20 pixels (`Width=10` o `Width=20`). Si cambiamos el valor de cualquiera de las propiedades, como podría ser incrementar el ancho del objeto de 10 a 20 pixels, ello representa una *odi i a ión en el estado*.

La diapositiva nos invita a imaginar el núcleo de un objeto como un conjunto de propiedades establecidas; vea cómo la identidad del objeto también forma parte del estado del mismo, pues **Name** es una propiedad, al igual que las de posición, apariencia, y otras.

www.detodoprogramacion.com

Comportamiento

El *o porta iento* es la capacidad del objeto para funcionar de una determinada manera. Como respuesta a la interacción del usuario, de otros objetos, o incluso del mismo sistema operativo, el objeto podrá comportarse de diferente forma.

El comportamiento de un objeto se presenta por la posibilidad de mandar llamar a ejecución:

- Los métodos.
- Los procedimientos de evento.
- Los procedimientos y funciones definidas por el usuario.

Los *todos*, como sabe, son los comportamientos predefinidos que un objeto puede presentar, y que se encuentran definidas por la clase de la cual el objeto deriva.

Los *pro edi ientos de e ento* son aquellos procedimientos que le suceden a un objeto, como respuesta a la interacción que tienen con el usuario o con otros objetos. El objeto tiene un conjunto de eventos que reconoce, y usted como desarrollador lo único que hace es decirle a su programa *(a)* que quiere que un objeto determinado sea sensible a los eventos que le sucedan (especificación `WithEvents` al momento de la declaración), *(b)* elaborar un procedimiento, y *(c)* especificarle a dicho procedimiento que será ejecutado en caso de que le suceda el evento a alguno de los objetos que maneje (*andles*).

En versiones anteriores de Visual Basic, un procedimiento de evento sólo podía servir para manejar las acciones que deberían suceder cuando a *un* objeto determinado le sucedía *un* evento determinado; de hecho, el nombre de los procedimientos de evento estaba formado por el nombre del objeto y el nombre de evento, separados por un guión bajo. Había controles para los cuales aplicaban las mismas validaciones, como por ejemplo que los cuadros de texto no se dejaran vacíos; no obstante que el proceso a realizar era el mismo para todos los cuadros de texto, a cada uno de ellos se le debía definir un procedimiento de evento.

Con Visual Basic.NET hay cosas que cambiaron. Un procedimiento de evento puede manejar un mismo evento para uno o más objetos; por ese motivo, el nombre asociado a un solo objeto y un solo evento ya no tiene sentido. Ahora el nombre del procedimiento de evento puede ser el que sea.

La única cosa que debemos respetar son los argumentos implícitos. a vimos en el módulo 13, que los procedimientos pueden admitir argumentos, y que con base en ellos, pueden modificar el resultado del procesamiento o el valor que retornan (en el caso de las funciones). Los *ar u entos i pl itos* son aquellos que son declarados por Visual Basic.NET de manera automática; en la declaración de este tipo de argumentos el

www.detodoprogramacion.com

programador no participa más que como espectador y usuario. Al conjunto de argumentos implícitos que requiere un procedimiento de evento, se le suele dar el nombre de *signature*.

Cada evento podrá exigir la presencia de los argumentos implícitos que necesita para trabajar; cuando nosotros declaramos un objeto especificando que queremos que sea sensible a eventos (`WithEvents`), también le estamos diciendo al programa que queremos que Visual Basic.NET se encargue de actualizar en tiempo de ejecución los datos de los argumentos implícitos de cada uno de los eventos aplicables al objeto. Nosotros no determinamos el valor que tendrán los argumentos implícitos; es el CLR, en tiempo de ejecución, quien determina qué valores tendrán.

Los *pro edi ientos un iones de inidas por el usuario* (*ser efined unctions*) son los procedimientos y funciones que usted codifica; no dependen de los eventos que pueden sucederle al objeto, ni requieren estar predefinidos en ninguna clase: son la forma más pura de codificación. Hace muchos años, cuando nadie hablaba aún de objetos ni de eventos, los procedimientos y funciones definidos por el usuario ya existían.

Los procedimientos y funciones definidas por el usuario son la parte más especializada de las aplicaciones, ya que contienen los bloques de código que han de realizar aquello que el lenguaje por sí mismo no puede. En ocasiones se dice que los procedimientos y funciones definidas por el usuario, son las *re las de ne ocio*, para la aplicación.

Todos los lenguajes tratados en este libro proporcionan sus propias bibliotecas de objetos, de entre los cuales se puede tomar lo que se necesite, pero son funcionalidades genéricas. Seguramente hay algún proceso o cálculo muy específico que nosotros deberemos desarrollar.

Por esa razón, es indispensable que sepamos desarrollar nuestros propios objetos, y desde luego, nuestras propias bibliotecas de clases.

Definición de una clase

Una recomendación es que, antes de que proceda a desarrollar una clase determinada, asegúrese de que no está desarrollada ya, y que forma parte de la biblioteca de clases del lenguaje que está utilizando; sería lamentable que después de que invierta tiempo y esfuerzo en el desarrollo de una clase se de cuenta de que ya existía desarrollado lo que usted necesitaba.

Si ya está seguro de que no existe una clase que se ajuste a lo que usted necesita, proceda entonces a generar una clase.

www.detodoprogramacion.com

Para crear una clase en Visual Basic.NET debe utilizar la siguiente sintaxis:

```
Class   o  bre  lase
   ódi o  lase
End Class
```

Donde o bre lase es el nombre a través del cual la clase puede ser invocada. La instanciación de la clase se deberá llevar a cabo por medio de este nombre. ódi o lase es el contenido de la clase; como sabemos, el contenido de la clase se compone de datos (propiedades) y comportamiento (métodos y funciones).

Por esa razón debemos definir dentro de la clase los procedimientos (`Sub` - `End Sub`), funciones (`Function` - `End Function`), y propiedades (`Property` - `End Property`) que tiene, entre otros elementos.

Una *lase*, como ya sabe, es la definición formal de un objeto. Las clases por sí mismas no pueden ser utilizadas. Es necesario crear unidades funcionales de las clases, es decir, objetos. Para ello debemos recurrir a un constructor, que es el código que genera la instancia de una clase. En Visual Basic.NET, un constructor típico sería:

```
Dim   b eto As New   o  bre  lase()
```

Donde *b eto* será el nombre de la instancia de la clase; *o bre lase* es el nombre de la clase que se está instanciando. `New` es la instrucción constructora por excelencia.

Una vez generada la instancia de la clase, podemos utilizar todos sus elementos, como podrían ser los métodos, las funciones y las propiedades.

Definición de propiedades

Una clase puede, además de definir su comportamiento a través de métodos y funciones, especificar qué propiedades que han de conformar su estado.

Las *propiedades* son las características particulares de un objeto; la clase debe definir qué propiedades tendrán los objetos que deriven de ellas. Para su definición, es necesario definir las propiedades dentro del bloque de código que compone la clase.

Las propiedades almacenan valores de un determinado tipo de dato, y las operaciones que se pueden hacer con dicho valor es leerlo (`GET`) o asignarlo (`SET`).

Las propiedades, aunque son valores almacenados, tienen un manejo más formal que las variables de memoria, y de hecho requieren de una variable de memoria que permite la verificación de los valores, antes de que sean leídos o asignados.

Sintaxis de la definición de propiedades:

```
Class   o  bre  lase
      Dim Variable As   ipo  ato  ropiedad
      Property   o  bre  ropiedad As   ipo  ato  ropiedad
            Get
                  ódi  o  e turaValor
            End Get
       Set (Variable  r  u  ento As   ipo  ato  ropiedad)
              ódi  o  si  na iónValor
       End Set
      End Property
      ...
      ...
End Class
```

Donde *o bre lase* es el nombre de la clase a la que pertenece la propiedad; antes de declarar la propiedad se debe declarar una variable de trabajo (*Variable*) del mismo tipo de dato que el que pretende almacenar la propiedad (*ipo ato ropiedad*).

Para recuperar el valor que tiene la propiedad se usa *ódi o e turaValor*, que es el bloque de código que se encarga de recuperar el valor de la variable de paso, para considerarlo como valor de la propiedad.

Para asignarle un valor a una propiedad se utiliza una variable de trabajo (*Variable r u ento*) que recibe, a manera de argumento, el valor que se pretende asignar la propiedad. *ódi o si na iónValor* es el bloque de código que se encarga de validar el valor que se pretende asignar a la propiedad; en caso de que sea válido, realiza la asignación.

Se pueden definir tantas propiedades como se requieran en el programa, y por supuesto, además se pueden definir métodos y funciones que formen parte de la misma. Estos métodos y funciones pueden hacer uso de las propiedades, con solo referirlas por su nombre.

Ejercicio 14.01

Creando una clase y generando una instancia

En este ejercicio se generará una clase, se generará una instancia (objeto) a partir de la clase, y se utilizará un método declarado en la clase.

1. Utilice un editor de texto plano (***Bloc de notas***, por ejemplo), para escribir el código fuente llamado Clases.vb; el archivo deberá guardarse en el directorio de trabajo del libro (C:\APIP).

Codificación de **Clases.vb**

```
1  Imports System
2
3  Module Clases
4
5        Class MiClase
6              Sub Mensaje()
7                     Console.Writeline ("Ejemplo de una clase")
8              End Sub
9        End Class
10
11       Sub Main()
12             ' Constructor de la clase derivada
13             Dim MiObjeto As New MiClase()
14             MiObjeto.Mensaje()
15             Console.Writeline ("Pulse INTRO para continuar")
16             Console.Readline()
17       End Sub
18
19 End Module
```

Ejecución guiada de **Clases.vb**

- Compile el programa **Clases.vb**, y ejecute el ensamblado que se genera.

  En un módulo, lo que se ejecuta es el procedimiento principal (**Main()**). En nuestro caso, el cuerpo del programa se describe en las líneas de la 12 a la 16.

  En la línea se comienza a declarar una clase, misma que termina en la línea ; todo lo que se encuentre entre la línea 6 y la , sólo podrá ser utilizado a través de una instancia de la clase.

En nuestro ejemplo, el único elemento que se incluye en la clase es un método (`Sub`).

¿Cuál es el nombre de la clase?

¿Cuál es el nombre del único método definido en la clase?

En la línea 13 se tiene un constructor. La instancia de la clase (objeto), se llama **MiObjeto**; parte por parte, la línea 13 declara un objeto (`Dim`) llamado **MiObjeto**, como (`As`) una nueva instancia (`New`) de la clase **MiClase**.

Después de la instanciación, todos los elementos contenidos en la clase serán accesibles a través del objeto **MiObjeto**.

La línea 14 ejecuta el método **Mensaje()**, que como podemos ver en la línea , sólo desplegará un mensaje que dice Ejemplo de una clase . Sólo a través de una instancia de la clase **MiClase**, es posible ejecutar la línea de código.

Si todo ocurrió como se esperaba, la salida será la siguiente:

```
Ejemplo de una clase
Presione INTRO para continuar
```

FIN DEL EJERCICIO ✱

Ejercicio 14.02

Definiendo y utilizando propiedades

En este ejercicio se generará una clase que define propiedades y las utiliza, a través de un objeto generado a partir de ella. El programa busca calcular el área de un cuadrado, utilizando para ello programación orientada a objetos.

1. Utilice un editor de texto y genere el programa `CalculaArea.vb`; guárdelo en su directorio de trabajo C: APIP.

Codificación de **CalculaArea.vb**

```
1  Imports System
2
3  Module CalculaArea
4
5      Class Cuadrado
6          Dim mBase As Decimal
7          Dim mAltura As Decimal
8
9          Property Base As Decimal
10             Get
11                 Return mBase
12             End Get
13             Set (Valor As Decimal)
14                     mBase = Valor
15             End Set
16         End Property
17
18         Property Altura As Decimal
19             Get
20                 Return mAltura
21             End Get
22             Set (Valor As Decimal)
23                     mAltura = Valor
24             End Set
25         End Property
26
27         Function Calculo() As Decimal
28             Return (Base * Altura)
29         End Function
30
31     End Class
32
33     Sub Main()
34         Dim MiObjeto1 As New Cuadrado()
35         Dim MiObjeto2 As New Cuadrado()
36
37         MiObjeto1.Base = 10
38         MiObjeto1.Altura = 20
39         Console.Writeline ("Area:" & MiObjeto1.Calculo())
40
41         MiObjeto2.Base = 15
42         MiObjeto2.Altura = 20
43         Console.Writeline ("Area:" & MiObjeto2.Calculo())
44
45         Console.Writeline ("Pulse INTRO para continuar")
46         Console.Readline()
47     End Sub
48
49 End Module
```

Ejecución guiada de **CalculaArea.vb**

2. Compile el programa `CalculaArea.vb`, y ejecute el ensamblado que se genera. Declaración de la clase.

   De la línea a la 31 se define una clase llamada **Cuadrado**, que tiene definidas 2 propiedades y un método. Las líneas 6 y declaran las variables de trabajo que las propiedades requerirán para poder recuperar un valor de una propiedad, o bien para asignarlo.

   De la línea a la 16 se declara la propiedad llamada **Base**, que se apoya para su funcionamiento en la variable de memoria declarada en la línea 6. El código de recuperación del valor de la propiedad está en la línea 11; como se puede ver, retorna el valor que actualmente tenga almacenado la propiedad, en la variable de memoria de trabajo utilizada por la propiedad. Cada propiedad deberá tener su propia variable de memoria de trabajo.

   Entre la línea 13 y la 1 se encuentra el código de asignación de valor a la propiedad. La línea 6 declara una variable de trabajo que almacena el valor que se pretende asignar a la propiedad.

   De la línea 1 a la 2 se declara otra propiedad llamada **Altura**, que se apoya para su funcionamiento en la variable de memoria declarada en la línea . La forma en que se realiza la recuperación de valores y su asignación, es igual a la forma que sucede con la propiedad **Base**.

   De la línea 2 a la 2 se declara una función (método) llamada **Calculo( )**; dicha función retornará un valor de tipo **Decimal** (línea 2), producto de multiplicar el contenido de la propiedad **Base**, por la propiedad **Altura** (línea 2).

- Proceso principal, instanciación, asignación de valores a las propiedades, y ejecución del método **Calculo( )**.

  En la línea 34 y 3 se tienen constructores que generan instancias de la clase **Cuadrado**. Por ser una instancia de **Cuadrado**, los objetos **MiObjeto1** y **MiObjeto2** ya poseen, cada uno por separado, sus propias propiedades **Base** y **Altura**, y sus métodos **Calculo( )**.

  En las líneas 3 y 3 se asignan valores a las propiedades del objeto **MiObjeto1**, y la línea 3 muestra en la consola el resultado de ejecutar el método **Calculo( )**, considerando los valores que en ese momento tengan las propiedades.

  En las líneas 41 y 42 se asignan valores a las propiedades del objeto **MiObjeto2**, y la línea 43 muestra en la consola el resultado de ejecutar el método **Calculo( )**, considerando los valores que en ese momento tengan las propiedades.

  Como puede comprobar, cada objeto mantiene los valores de sus propiedades.

Si todo ocurrió como se esperaba, la salida será la siguiente:

```
Area:200
Area:300
Pulse INTRO para continuar
```

FIN DEL EJERCICIO ✱

Herencia

La *eren ia* (*in eritan e*) es un concepto fundamental de la programación orientada a objetos, y consiste en la habilidad de definir clases que sirven como base de otras clases, evitando la redundancia en la definición de elementos tales como métodos, eventos, y propiedades. Tenemos una clase base y una o varias clases derivadas; las clases derivadas aprovechan todo lo declarado en la clase base, asumiéndolo como suyo y utilizando capacidades que no están codificadas en ellas.

Las clases derivadas pueden heredar, e incluso extender, las propiedades, métodos y eventos de la clase base. Es posible también que las clases derivadas sustituyan o modifiquen (*o erride*) la funcionalidad de los métodos de la clase base.

En Visual Basic.NET, por omisión todas las clases pueden heredarse.

Las ventajas de la herencia son muchas, pero la principal es que el desarrollador sólo tendrá que escribir y depurar el código una vez, para una clase; no obstante a ello, podrá disponer en muchas clases la misma funcionalidad, estableciendo las referencias adecuadas.

¿Cuándo se debe utilizar la herencia?

La herencia entre clases no debe utilizarse por el simple hecho de que el lenguaje lo permite. La herencia debe planearse; el desarrollador debe pensar cómo han de ser las clases, organizándolas en lo que se conoce como *erar u a de lases*.

Se recomienda una jerarquía de clases cuando entre las clases se presenta una relación de tipo *Es un* (*is a*); debe evitarse el uso de herencia cuando se trata de relaciones *iene un* (*as a*). Una clase que contiene las características de los automóviles en general (`Automóviles`), tiene mucho que compartirle a una clase que almacena las particularidades de los automóviles deportivos (`AutomóvilesDeportivos`), pues un automóvil deportivo, antes de ser deportivo, es un automóvil (*is a*); ahora pregúntese, ¿qué comparte un dueño de un automóvil y un automóvil? `Automóviles` y `DueñosAutomóviles`, no comparten rasgos, ni siquiera similares. Las personas que poseen automóviles no se derivan de los automóviles, ni viceversa. Que una persona tenga un (*as a*) automóvil, no quiere decir que comparta con éste los métodos,

propidades y eventos. Imagínese tener que determinar el tipo de transmisión, número de puertas, y número de serie que una persona posee. Simplemente, de una cosa no puede derivar la otra.

Se recomienda una jerarquía de clases, obviamente, sólo si el código de una clase brinda alguna funcionalidad que se pueda reutilizar; tratar de heredar aspectos muy particulares que sólo aplican para una clase y no para otras no tiene sentido.

Inherits (Heredar)

El estatuto `Inherits` se utiliza para especificar que una clase de nueva creación tiene como base una clase ya existente. La clase nueva es la *lase deri ada* (*deri ed lass*); la clase ya existente es la *lase base* (*base lass*).

La sintaxis es la siguiente:

```
Class o bre lase eri ada
      Inherits o bre lase ase
      ...
End Class
```

En este caso, la clase derivada dispone de todos los elementos declarados en la clase base.

NotInheritable (No Heredable)

a decíamos que por omisión todas las clases en Visual Basic.NET pueden heredarse. Existirán ocasiones en que no queramos que una clase pueda heredarse. En ese caso, hay que especificar, al declarar una clase, que ésta no es heredable.

La sintaxis es la siguiente:

```
NotInheritable Class o bre lase
      ...
End Class
```

En este caso, intentar heredar *o bre lase* generaría error (`Inherits NombreClase`).

MustInherit (Debe Heredar)

Como sabemos, los objetos son instancias de una clase. a explicábamos que no se puede tener acceso a la funcionalidad de una clase, sino a través de una instancia de la misma, es decir, de un objeto basado en la clase. Eso es verdadero sólo relativamente.

¿Habrá alguna forma de que podamos disponer de la funcionalidad de una clase, sin generar una instancia de la misma?

La respuesta es que sí. Podemos declarar una clase que no pueda instanciarse, y que pueda ser utilizada sólo si se hereda por otra clase. Para hacer eso es necesario especificar el estatuto `MustInherit`.

La sintaxis es la siguiente:

```
MustInherit Class   o  bre  lase
        ...
End Class
```

En este caso, instanciar *o bre lase* generaría error (`Dim Objeto As New NombreClase()`)

Modificadores de acceso, necesarios para la herencia

Usted debe determinar el alcance que tendrá su jerarquía de clases, a efecto de restringir el acceso a las clases, de tal manera que sólo puedan heredarlas aquellas porciones de código que deban hacerlo.

Calificar una clase como *lase p bli a* (*ubli lass*), se logra utilizando el modificador de acceso `Public`, al momento de la declaración de la clase. Esto garantiza que todas las clases podrán utilizar la clase todo el tiempo; en este caso, el acceso es abierto.

Si no quiere una clase de acceso abierto, puede restringir el acceso a la clase utilizando los modificadores de acceso `Friend` y `Private`.

Tome en cuenta que en una jerarquía de clases el acceso sólo puede heredarse hacia un acceso igual o mayor, nunca menor; una clase con acceso `Public` no puede heredar una clase `Friend` o `Private`, y una clase `Friend` no puede heredar una `Private`.

Estatutos auxiliares de herencia

En ocasiones heredamos una clase, pero nos damos cuenta que algunas entidades (métodos, propiedades, eventos) no se ajustan del todo a lo que verdaderamente necesitamos; en resumen, no nos sirven tal y como fueron heredadas. Es posible que pensemos que los elementos heredados son útiles, pero lo son más si podemos hacer unas pequeñas modificaciones.

Dichas modificaciones se realizan en la clase derivada, respecto a las definiciones originales existentes en la clase base.

Estatutos auxiliares de herencia

La programación orientada a objetos hace posible que eso suceda, pues permite implementar diferentes comportamientos para un método (*o erride*), o de plano remplazarlo por completo (*o erload*). La funcionalidad heredada puede modificarse por otra que sea más indicada para el problema de programación que deseamos resolver; con ello se promueve que la reutilización del código sea flexible y adaptable.

Por omisión, una clase derivada hereda los métodos y propiedades de su clase base. Si necesitamos que una propiedad o método heredado se comporte de manera distinta a como se definió en la clase base, puede ser implementada de forma distinta (*o erridden*) o remplazada (*o erloaded*); esto quiere decir que la herencia sirve hasta donde nosotros deseamos comenzar a hacer modificaciones.

Los *estatutos auxiliares de eren ia* sirven para especificar el grado de modificación que ha de darse entre la clase derivada y la clase base; estos son: `Overridable`, `NotOverridable`, `Overrides`, `MustOverride`, y `Overloads`.

Del lado de la clase base hay tres escenarios.

NotOverridable (No Remplazable)

El primer escenario es que la clase base no acepte que su funcionalidad sea modificada o sustituida. En ese caso, el estatuto `NotOverridable` previene que una propiedad o método de una clase base pueda ser sustituido en una clase derivada. Por omisión, los métodos públicos en una clase base son `NotOverridable`.

Overridable (Remplazable)

El segundo escenario es que la clase base admita modificaciones o sustituciones a su funcionalidad. El estatuto `Overridable` establece que una propiedad o método (elemento) de una clase base puede ser modificado en su implementación o sustituido en una clase derivada.

MustOverride (Debe Remplazarse)

El tercer escenario es que la modificación a la funcionalidad de la clase base no sólo sea posible, sino que sea obligatoria. El estatuto `MustOverride` hace que sea obligatorio hacer variaciones en la clase derivada, de las propiedades o método de la clase base, que han sido heredados.

Del lado de la clase derivada hay dos escenarios, que sólo son posibles si los elementos de la clase base permiten modificaciones en la funcionalidad.

Overrides (Remplaza)

El primer escenario es cuando la clase base permite que su funcionalidad sea modificada, respetando la estructura original del elemento modificado. El estatuto `Overrides` permite la variación en la clase derivada, de sus propiedades o sus métodos. Esto será siempre y cuando se mantengan los mismos argumentos en el método de la clase base y de la clase derivada. `Overrides` sólo permite desarrollar en la clase derivada una implementación distinta de un elemento de la clase base, y no su sustitución por otro que no guarde semejanza.

Overloads (Sobrecarga/Sustituye)

El segundo escenario es que se desee sustituir la funcionalidad, sin necesidad de respetar estructuras o condiciones precedentes.

En ocasiones además de modificar el comportamiento de los métodos que participan en una relación de herencia, es necesario hacer uso de diferentes argumentos, ya sea porque los que se tienen deben considerar otros tipos de datos, o bien, que el método que modifica al heredado de la clase base requiere una cantidad distinta de argumentos. En este caso, ya no hablamos de una implementación distinta de lo mismo, sino de dos métodos bastante distintos entre sí, que comparten un nombre.

En este último caso, `Overrides` no sirve, pues exige igualdad de argumentos. Para poder lograr nuestro cometido, debemos utilizar `Overloads`, que indica la sustitución del método de la clase base por otro de la clase derivada, tengan o no relación entre sí. A través de `Overloads` es la única forma de implementar, bajo un mismo nombre, dos métodos que no coinciden más que en el nombre.

MyBase y MyClass

Se puede utilizar la palabra reservada `MyBase`, para llamara a ejecución a un método de la clase base, desde una clase derivada. Esto quiere decir que, *a* si existe herencia entre dos clases, y en la clase derivada se modifica un método, entonces *c* es posible mandar llamar a ejecución al método, ya sea en la forma en que se encuentra en la clase base, o en la forma en que se encuentra en la clase derivada misma.

Con `MyBase` podemos obligar a la aplicación a que utilice un método de la clase base y no de la clase derivada; caso contrario, es posible que queramos mandar llamar el método tal y como está en la clase derivada. Para ello utilizamos la palabra reservada `MyClass`, que se encarga de llamar a ejecución un método que se encuentra en la misma clase, cuando esta es derivada.

Es importante saber que tanto `MyBase` como `MyClass`, aunque en su manejo parecen un objeto, realmente no lo son. No se instancian; existen porque el lenguaje los proporciona para un manejo más flexible de la herencia y el polimorfismo.

www.detodoprogramacion.com

Polimorfismo

El *poli or is o* es la habilidad que tiene el lenguaje para definir múltiples clases que contengan métodos y propiedades del mismo nombre, pero que brindan diferente funcionalidad.

Cuando se presenta el polimorfismo es posible elegir, en tiempo de ejecución, aquellos métodos o propiedades que necesitamos. No importa que varias clases posean los mismos métodos y las mismas propiedades: la aplicación debe ser capaz de utilizar precisamente aquellas cuya funcionalidad estamos invocando.

La herencia y la sustitución de métodos y propiedades tienen mucho que ver con el polimorfismo. Cuando se declara una clase y a partir de ésta se derivan otras clases, estamos ante la presencia de la herencia; si además de esto, los métodos y propiedades de la clase base que fueron heredados son modificados (*erriden*) o sustituidos (*erloaded*) en la clase derivada, de tal manera que comparten el mismo nombre pero no la misma funcionalidad, estamos ante el polimorfismo. La manifestación, aun más pura del polimorfismo se presenta cuando se utiliza `Overloads`, que implica una sustitución completa de métodos que se encuentran en clases que guardan una relación de herencia.

Cuando en una clase derivada se modifican métodos o propiedades de la clase base, se dice que hay *poli or is o basado en eren ia*.

Ejercicio 14.03

Creando una jerarquía de clases por herencia

En este ejercicio se generará una clase base y una clase derivada; se comprobará la herencia entre ellas.

1. Utilice un editor de texto y genere el programa `Herencia.vb`; guárdelo en su directorio de trabajo C: APIP.

Codificación de **Herencia.vb**

```
1  Imports System
2
3  Module Herencia
4
```

```
5         Class MiClaseBase
6               Sub Mensaje1()
7                     Console.Writeline ("Clase base, mensaje 1")
8               End Sub
9               Sub Mensaje2()
10                    Console.Writeline ("Clase base, mensaje 2")
11              End Sub
12        End Class
13
14        Class MiClaseDerivada
15              Inherits MiClaseBase
16              Sub Mensaje3()
17                    Console.Writeline ("Clase derivada, mensaje 3")
18              End Sub
19        End Class
20
21         Sub Main()
22              ' Constructor de la clase derivada
23              Dim MiObjeto As New MiClaseDerivada()
24              MiObjeto.Mensaje1()
25              MiObjeto.Mensaje2()
26              MiObjeto.Mensaje3()
27              Console.Writeline ("Pulse INTRO para continuar")
28              Console.Readline()
29        End Sub
30
31  End Module
```

Ejecución guiada de **Herencia.vb**

1. Compile el programa **Herencia.vb**, y ejecute el ensamblado que se genera.

   En un módulo, lo que se ejecuta es el procedimiento principal (**Main()**). En nuestro caso, el cuerpo del programa se describe en las líneas de la 22 a la 30.

   El módulo completo consta de 2 entidades de tipo clase (**MiclaseBase**, de la línea a la 12, y **MiClaseDerivada**, de la línea 14 a la 1 . Además, tiene un procedimiento principal (**Main()**)

   La clase **MiClaseBase** tiene dos entidades de tipo método (**Mensaje1( )**, de la línea 6 a la , y **Mensaje2( )**, de la línea a la 11). La clase **MiClaseDerivada** tiene un solo elemento de tipo método (**Mensaje3( )**, de la línea 16 a la 1)

   Tome en cuenta que la clase **MiClaseDerivada** no posee los métodos **Mensaje1( )** y **Mensaje2( )**, pues la que los posee es la clase **MiClaseBase**.

La clase **MiClaseDerivada** hereda la clase **MiClaseBase** (línea 1); aun y cuando **Mensaje1( )** y **Mensaje2( )** no están declaradas en **MiClaseDerivada**, ya forman parte de dicha clase, debido a que son tomadas de la clase base.

Todos los métodos muestran un mensaje en la consola, que indica que se ejecutaron.

- Utilizando métodos heredados.

En la línea 23 se tiene un constructor, que genera una instancia de la clase **MiClaseDerivada**, bajo el nombre de **MiObjeto**. Vea cómo las líneas 24 y 2 hacen uso de métodos no declarados en la clase **MiClaseDerivada**; esto es posible debido a que por haber establecido una relación de herencia con **MiClaseBase**, la clase derivada automáticamente dispone de *todos* los métodos, propiedades y eventos declarados en la clase base.

De esta forma, **MiClaseDerivada** sólo declarará cosas que no están declaradas en **MiClaseBase**; declarar los mismos métodos en ambas clases sería redundante.

Tome en cuenta las dimensiones que puede tomar su jerarquía de clases, ya que si hereda una clase que a su vez es derivada de otra clase, se hereda también la funcionalidad de la clase base de ésta.

Si todo ocurrió como se esperaba, la salida será la siguiente:

```
Clase base, mensaje 1
clase base, mensaje 2
Clase derivada, mensaje 3
Presione INTRO para continuar
```

FIN DEL EJERCICIO ✱

Ejercicio 14.04

Comprobando el polimorfismo basado en herencia, y el uso de MyBase

En este ejercicio se comprobará el polimorfismo basado en herencia; para ello se definirán varias clases, se heredarán unas con otras, y se verá el uso de la palabra reservada `MyBase`. El programa calculará lo que se le pagará a los empleados, partiendo del supuesto que puede haber pago normal, agregarse un bono por puntualidad del 2 , e incluso un bono de extra del 20 .

1. Utilice un editor de texto y genere el programa `Nomina.vb`; guárdelo en su directorio de trabajo C: APIP.

Codificación de **Nomina.vb**

```
1  Imports System
2
3  Module Nomina
4
5        Const PagoXHoraVig As Decimal = 120
6        Const BonoPuntualidad As Decimal = 1.25
7
8        Class Pago
9              Overridable Function CalculaPago( _
10                    ByVal HorasTrabajadas As Decimal, _
11                    ByVal PagoXHora As Decimal) As Decimal
12                    Return HorasTrabajadas * PagoXHora
13             End Function
14       End Class
15
16       Class PagoBonificado
17       ' Se define la clase Pago como clase base de PagoBonificado
18           Inherits Pago
19           Overrides Function CalculaPago( _
20                    ByVal HorasTrabajadas As Decimal, _
21                    ByVal PagoXHora As Decimal) As Decimal
22             Return MyBase.CalculaPago(Horastrabajadas, PagoXHora) * _
23                    BonoPuntualidad
24           End Function
25       End Class
26
27       Class PagoBonificadoExtra
28       ' Se define la clase Pago como clase base de PagoBonificadoExtra
29           Inherits Pago
30           Overloads Function CalculaPago( _
31                  ByVal HorasTrabajadas As Decimal, _
32                  ByVal PagoXHora As Decimal, _
33                  ByVal Extra As Decimal) As Decimal
34             Return MyBase.CalculaPago(Horastrabajadas, PagoXHora) * _
35                    BonoPuntualidad * Extra
36          End Function
37       End Class
38
39       Sub Main()
40             Dim PagoNoPuntuales As Pago = New Pago()
41             Dim PagoPuntuales As New PagoBonificado()
42             Dim PagoPuntualesPlus As New PagoBonificadoExtra()
43             Dim Horas As Decimal = 40
44             Console.Writeline ("Pago a los no puntuales: " & _
45                PagoNoPuntuales.CalculaPago(Horas, PagoXHoraVig))
46             Console.Writeline ("Pago a los puntuales: " & _
47                PagoPuntuales.CalculaPago(Horas, PagoXHoraVig))
48             Console.Writeline ("Pago a los puntuales, + extra: " & _
49                PagoPuntualesPlus.CalculaPago(Horas, _
50                    PagoXHoraVig, 1.2))
```

```
51              Console.Writeline ("Pulse INTRO para continuar")
52              Console.Readline()
53        End Sub
54
55  End Module
```

Ejecución guiada de **Nomina.vb**

1. Compile el programa `Nomina.vb`, y ejecute el ensamblado que se genera. Estructura general del programa.

   Al inicio del programa se definen dos constantes que serán utilizadas en los cálculos; **PagoXHoraVig**, que es la cantidad vigente que se puede pagar a un trabajador por hora laborada (línea), y **BonoPuntualidad**, que es el factor que se aplicará al salario que deba pagarse al trabajador normalmente, en caso de que haya sido puntual todos los días (línea 6)

   En un módulo, lo que se ejecuta es el procedimiento principal (**Main( )**). En nuestro caso, el cuerpo del programa se describe en las líneas de la 3 a la 4.

   El módulo completo consta, además de **Main( )** , de 3 clases: **Pago**, **PagoBonificado**, y **PagoBonificadoExtra**.

   La clase **Pago** actúa como clase base de **PagoBonificado** y **PagoBonificadoExtra**, pues el pago que se realiza a los trabajadores siempre deriva del pago normal, al cual se le agregan diferentes conceptos de afectación.

- Comprobando la herencia con sustitución de elementos.

  La clase **Pago** actúa como clase base de **PagoBonificado** y **PagoBonificadoExtra**, que son derivadas. La clase **Pago** contiene un único método, llamado **CalculaPago( )**, que requiere dos argumentos para realizar sus cálculos: *(a)* uno de tipo **Decimal** llamado **HorasTrabajadas**, en donde se proporcionará el número de horas que el trabajador haya laborado (línea 10), y *(b)* otro de tipo Decimal llamado **PagoXHoras**, en donde se proporcionará el pago por hora que se considerará para el cálculo del salario (línea 11).

  El método producirá un resultado de tipo **Decimal**, que resultará de multiplicar los dos argumentos proporcionados (línea 12). La multiplicación de las horas trabajadas y el pago por horas es la base del cálculo del salario, independientemente si hay pagos adicionales o no; considerando eso, todo pago que exista, al menos tiene estos elementos, y por tanto todo pago realizado es una (*s a*) derivación de este cálculo.

Es muy importante notar que el método **CalculaPago( )** está siendo definido como sustituible (`Overridable`), lo que permitirá que clases derivadas de la clase **Pago** podrán generar, en sustitución, nuevas implementaciones del método.

La clase **PagoBonificado** en esencia hace lo mismo que la clase **Pago**, con la diferencia que afecta el salario normal, aplicándole un 2 más, por concepto de puntualidad (líneas 22 y 23). **PagoBonificado** hereda métodos y propiedades de la clase **Pago** (línea 1); en virtud de que el cálculo que hace **PagoBonificado** tiene variaciones con respecto al realizado en pago (por agregarle 2 al salario), el método heredado **CalculaPago( )** no es útil tal como está, y es necesario una nueva implementación de dicho método.

En la línea 1 se declara el método **CalculaPago( )**, en que implementa de manera distinta al declarado en la clase base (línea). El método **CalculaPago( )** de la clase base puede ser sustituido, en virtud de que está calificado como *erridable*; se re codifica el método **CalculaPago( )**, especificándose que actuará en sustitución (`Overrides`) del método heredado de la clase base (línea 1).

- Comprobando el uso de **MyBase**.

Vea cómo en la línea 34 se hace referencia a **MyBase.CalculaPago()**; la razón de esto es que lo que ha de calcular **CalculaPago( )** de la clase **PagoBonificado** es lo mismo que aplicarle el factor de bono de puntualidad a lo calculado por el método **CalculaPago( )** de la clase **Pago**. En ese sentido, solicitamos que se ejecute **CalculaPago( )** de la clase base (**MyBase**), y que al resultado que se obtenga se le aplique el factor de bono de puntualidad. En caso de que no se especificara **MyBase**, lo que el programa trataría de hacer sería ejecutar el método **CalculaPago( )** que se encuentre en la misma clase, lo que sería imposible, pues **CalculaPago( )** de la clase **PagoBonificado** no se puede llamar a ejecución a sí misma, pues se generaría un ciclo infinito de ejecución. El problema queda resuelto, pues se llama a ejecución el método que se encuentra en la clase base (**MyBase**), y no el que se encuentra en la misma clase (**MyClass**)

- Sustitución de métodos con diferentes argumentos, usando `Overloads`.

La clase **PagoBonificadoExtra** contiene un método llamado **CalculaPago( )**; dicho método no comparte los mismos argumentos con el mismo método en la clase base, pues agrega uno de tipo **Decimal**, llamado **Extra**, que es un factor que se le aplica al sueldo, aún después del bono de productividad (línea 3)

La diferencia de argumentos de **CalculaPagos( )** de **PagoBonificadoExtra** nos dice que es posible que haga cosas distintas al mismo método en la clase base. Por ese motivo, no puede ser sustituido utilizando `Overrides`, sino `Overloads` (línea 30), pues la sustitución es total.

- Polimorfismo.

  En las líneas 40, 41 y 42 se generan instancias de cada una de las clases del programa.

  Las líneas de la 4 a la 1 se encargan de ejecutar un mismo método llamado **CalculaPagos( )** , y mostrar los resultados en la consola; los resultados correspondientes a cada método van en función a la clase a partir de la cual se generó cada objeto que manda llamar el método.

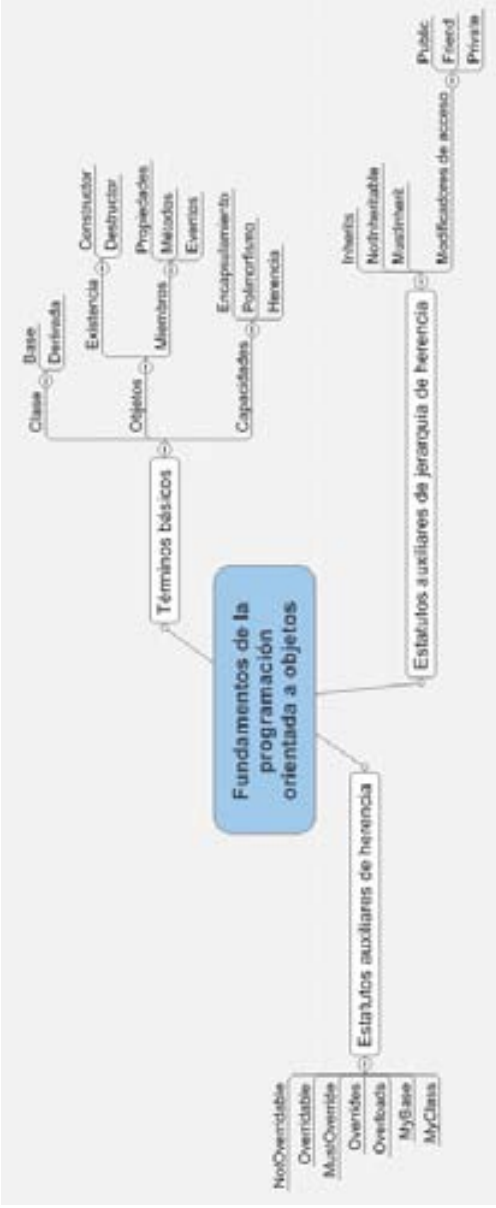
  El hecho de que bajo un mismo nombre de método se produzcan diferentes resultados, nos lleva a comprobar el polimorfismo.

  Si todo ocurrió como se esperaba, la salida será la siguiente:

```
Pago a los no puntuales: 4800
Pago a los puntuales: 6000
Pago a los puntuales, + extra: 7200
Pulse INTRO para continuar
```

FIN DEL EJERCICIO ✱

Mapa mental del capítulo



Terminología

Argumentos implícitos, 409
Base class, 405
Clase, 403
Clase base, 405
Clase derivada, 405
Clase pública, 419
Comportamiento, 409
Constructor, 403
Derived class, 405
Destructor, 403
Encapsulamiento, 404
Estatutos auxiliares de herencia, 419
Eventos, 404
Friend, 419
Herencia, 405, 417
Identidad, 407
Inheritance, 405, 417
Inherits, 418
Instancia, 403
Instanciación, 403
Members, 404
Métodos, 404
Miembros, 404
Modularidad, 404
Módulo, 404
MustInherit, 418
MustOverride, 420
MyBase, 421
MyClass, 421
NotInheritable, 418
NotOverridable, 420
Objeto, 403
Overloads, 421
Overridable, 420
Overrides, 421
Polimorfismo, 406, 422
Polimorfismo basado en herencia, 422
Private, 419
Procedimiento de evento, 409
Procedimientos y funciones definidas por el usuario, 410
Programación orientada a objetos, 403
Properties, 404
Propiedad de identidad, 408
Propiedades, 404, 411
Reglas de negocio, 410
Signature, 410
UDF, 410
Unidad funcional, 404
User defined functions, 410

Preguntas

14.1 ¿Cuáles son las ventajas de la programación orientada a objetos respecto a otros modelos de programación?

14.2 Cuáles son las tres características que distinguen a un lenguaje orientado a objetos. Explíquelas.

14.3 Mencione las diferencias fundamentales entre una clase y un objeto.

14.4 Mencione las diferencias fundamentales entre una clase base y una derivada.

Examen rápido del módulo

Parte 1: Marque con una cruz el inciso que considere que responde a la pregunta.

A la acci n e enerar n ob eto a artir e na clase se le lla a
(a) Instanciación
(b) Generación de objetos
(c) Derivación

Es el roce i iento o c i o e libera rec rsos e a no tili ar n ob eto
(a) Constructor
(b) Destructor
(c) GarbageCollector

Es la abili a e crear na n e a clase basa a en na clase e istente
(a) Polimorfismo
(b) Herencia
(c) Encapsulamiento

Es la ca aci a e tiene n ob eto e al acenar atos roce i ientos co o na ni a ncional sin e los e s ob etos eban conocerlos
(a) Polimorfismo
(b) Herencia
(c) Encapsulamiento

on las car cter sticas artic lares e los ob etos
(a) Propiedad
(b) Método
(c) Evento

Parte 2: Coloque en la línea la letra «F» si la sentencia es falsa, y «V» si la sentencia es verdadera.

6) La clase derivada hereda su funcionalidad a la clase base, en la herencia.

) El estado de un objeto es su capacidad de tener un nombre que lo identifica y lo diferencia de los demás objetos.

) Los objetos que producirán la ejecución de procedimientos de evento requieren ser creados con la cláusula ithEvents

) En programación POO, la codificación redundante es necesaria.

10) Los métodos son comportamientos predefinidos de un objeto.

Anexo 1:

Casos resueltos

Contenido

Caso 1: Mensaje 434
Caso 2: Niveles de servicio 439
Caso 3: Muestra de una tabla de multiplicar 448
Caso 4: Muestra las tablas de multiplicar del 1 al 5 456
Caso 5: Análisis de promedios 464

En el presente anexo se muestra la resolución completa de los casos prácticos del libro.

www.detodoprogramacion.com

Caso 1: Mensaje

Este programa tiene como finalidad comprobar técnicas y herramientas de desarrollo. Lo que el programa debe hacer es preguntar por un mensaje de saludo, para posteriormente mostrar el mensaje que se le proporcionó.

Utilidad del programa:

1. Comprueba la estructura básica de los programas en cualquier lenguaje.
2. Comprueba los estatutos de asignación de datos.
3. Comprueba los estatutos de recopilación de datos.
4. Comprueba los estatutos para el desplegado de datos existentes.

Análisis del caso

Área del conocimiento: Ninguna.

Análisis de los elementos del todo:

1. Se debe preguntar un mensaje.
2. Se debe mostrar el mensaje.

Unidades de funcionalidad

ID	Unidad de funcionalidad	Prioridad	Obligatoriedad
1	Preguntar un mensaje, para luego desplegarlo.	Alta	Si

Interfaz

Escribe un mensaje a mostrar: _____

<Aquí se muestra el mensaje>

Algoritmo

1. Inicio.
2. Se pregunta un mensaje.
3. Se muestra el mensaje.
4. Fin.

Simulación

El programa se comportaría de la siguiente manera si le proporcionamos el mensaje "Hola Mundo".

```
Escribe un mensaje a mostrar: Aprenda
Aprenda
```

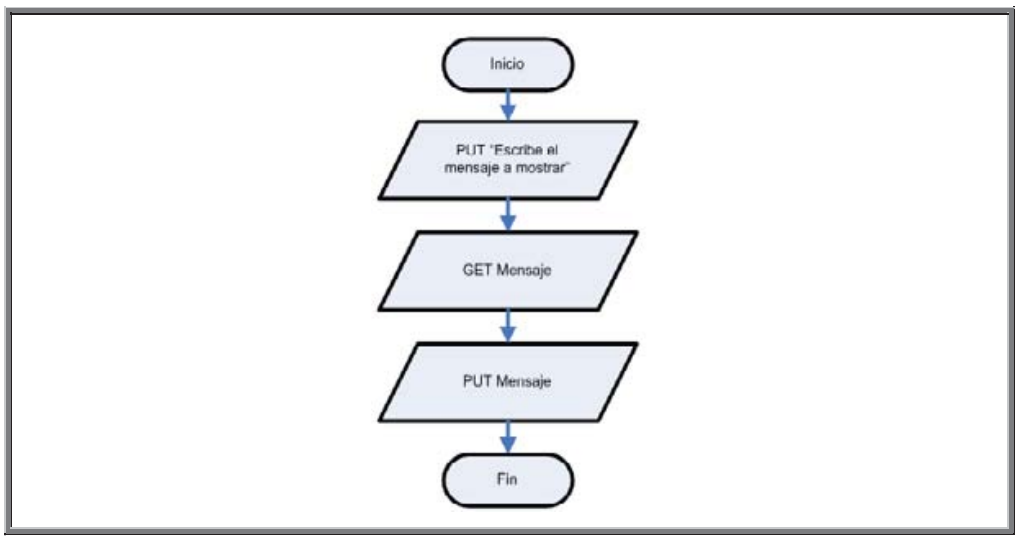
Algoritmo orientado a datos

1. Inicio.
2. Se pregunta un mensaje (Mensaje, x(30)) .
3. Se muestra el mensaje (Mensaje).
4. Fin.

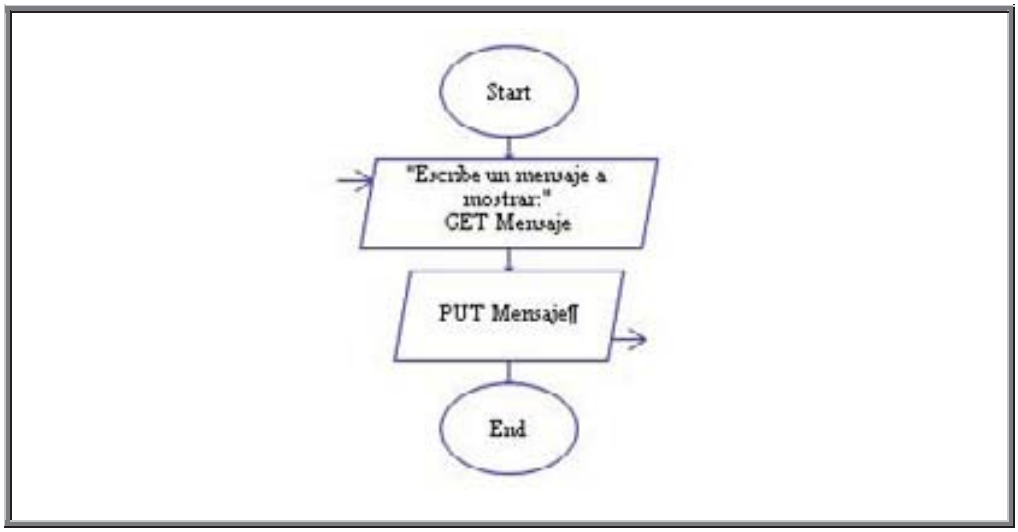
Miniespecificación

```
1   INICIO:
2           (Mensaje, x(30))
3           « "Escribe un mensaje a mostrar: "
4           » Mensaje
5           « Mensaje
6   :FIN
```

Diagrama de flujo del programa



Programa en Raptor



Codificación de **C1Mensaje.vb**

```
1   Imports System
2   Imports System.Console
3
4   Module C1Mensaje
5       Sub Main()
6
7           Dim Mensaje As String
8           Write("Escribe un mensaje a mostrar: ")
9           Mensaje = ReadLine()
10          WriteLine(Mensaje)
11
12      End Sub
13  End Module
```

Codificación de **C1Mensaje.cs**

```
1   using System;
2
3   class C1Mensaje
4   {
5           static void Main()
6           {
7
8                   String Mensaje;
9                   Console.Write("Escribe un mensaje a mostrar: ");
10                  Mensaje= Console.ReadLine();
11                  Console.WriteLine(Mensaje);
12
13          }
14  }
```

Codificación de **C1Mensaje.cpp**

```
1   #include <iostream.h>
2   using namespace std;
3
4   void main()
5   {
6
7           string Mensaje;
8           cout << "Escribe un mensaje a mostrar: ";
9           cin >> Mensaje;
10          cout <<  Mensaje << endl;
11
12  }
```

Codificación de **C1Mensaje.java**

```
1   import java.io.*;
2   import java.lang.*;
3
4   class C1Mensaje
5   {
6           public static void main(String args[])
7                   throws IOException
8     {
9
10                  BufferedReader entrada = new
11                          BufferedReader(new InputStreamReader(System.in));
12
13                  String Mensaje;
14                  System.out.print("Escribe un mensaje a mostrar: ");
15                  Mensaje=entrada.readLine();
16                  System.out.println(Mensaje);
17
18          }
19  }
```

Caso 2: Niveles de servicio

Se desea saber la disponibilidad del servicio de Internet para una persona, expresado en horas de disponibilidad.

Dicho nivel de servicio depende del grado de disponibilidad que ofrece la computadora que utiliza, así como del grado de disponibilidad del enlace a Internet que utiliza la máquina.

El horario de labores de la persona que los utiliza la computadora y el enlace a Internet es de 8 horas.

Para poder realizar los cálculos adecuadamente, tome en cuenta que la disponibilidad de un sistema de dispositivos (dos o más dispositivos que trabajan de forma conjunta) se calcula multiplicando los porcentajes de grado de disponibilidad de cada uno de ellos, y el resultado siempre será menor a la disponibilidad del menor.

Para calcular el tiempo de disponibilidad de un servicio (en este caso el acceso a la información publicada en Internet), sólo basta con multiplicar el tiempo en que es requerido el servicio por la disponibilidad del sistema de dispositivos.

Utilidad del programa:

1. Comprueba la realización de cálculos para la generación de datos nuevos.

Análisis del caso

Área del conocimiento: Cálculo de niveles de servicio.

Análisis de los elementos del todo:

1. Se desea conocer la disponibilidad del servicio de Internet, expresado en horas.
2. El grado de disponibilidad de la computadora influye en el grado de disponibilidad del servicio.
3. El grado de disponibilidad del servicio de enlace a Internet influye en el grado de disponibilidad del servicio.
4. El tiempo total en el que se requiere el servicio es de 8 horas.
5. La disponibilidad de un sistema de dispositivos se calcula multiplicando los porcentajes del grado de disponibilidad de los mismos.
6. La computadora y el servicio de enlace a Internet trabajan como un sistema de dispositivos.

Unidades de funcionalidad

ID	Unidad de funcionalidad	Prioridad	Obligatoriedad
1	Calcular el tiempo de disponibilidad del servicio, en horas, y mostrar el resultado.	Alta	Si

Interfaz

Grado de disponibilidad de la computadora: _____

Grado de disponibilidad del enlace: _____

Tiempo laborable en horas: _____

La disponibilidad del servicio de Internet es de <resultado> horas.

Algoritmo

1. Inicio.
2. Se pregunta la disponibilidad de la computadora
3. Se pregunta la disponibilidad del enlace
4. Se pregunta el tiempo laborable en horas
5. Se calcula la disponibilidad del sistema de dispositivos
6. Se calcula el tiempo de disponibilidad del servicio
7. Se muestra el tiempo de disponibilidad del servicio
8. Fin.

Simulación

Supongamos que la computadora que la persona utiliza brinda una disponibilidad del 98%, es decir, el 98% del tiempo funciona. Por otro lado, se ha comprobado que el enlace a Internet de banda ancha que la persona utiliza brinda una disponibilidad del 87%, es decir, el 87% del tiempo se tiene comunicación.

Disponibilidad del sistema =

Disponibilidad de la computadora * Disponibilidad del enlace

Aquí un ejemplo de cómo se realizarían los cálculos.

DS = DC * DE

DS = 0.98 * 0.87

DS = 0.85

La disponibilidad del sistema sería del 85.26%. La disponibilidad en horas del servicio sería el 85.26% del tiempo laborable.

Disponibilidad en horas = Disponibilidad del sistema * Tiempo laborable

DH = DS * TL

DH = 0.8526 * 8

DH = 6.82

La respuesta del programa sería la siguiente:

```
Grado de disponibilidad de la computadora: 0.98
Grado de disponibilidad del enlace: 0.87
Tiempo laborable en horas: 8
La disponibilidad del servicio de Internet es de 6.82 horas.
```

Algoritmo orientado a datos

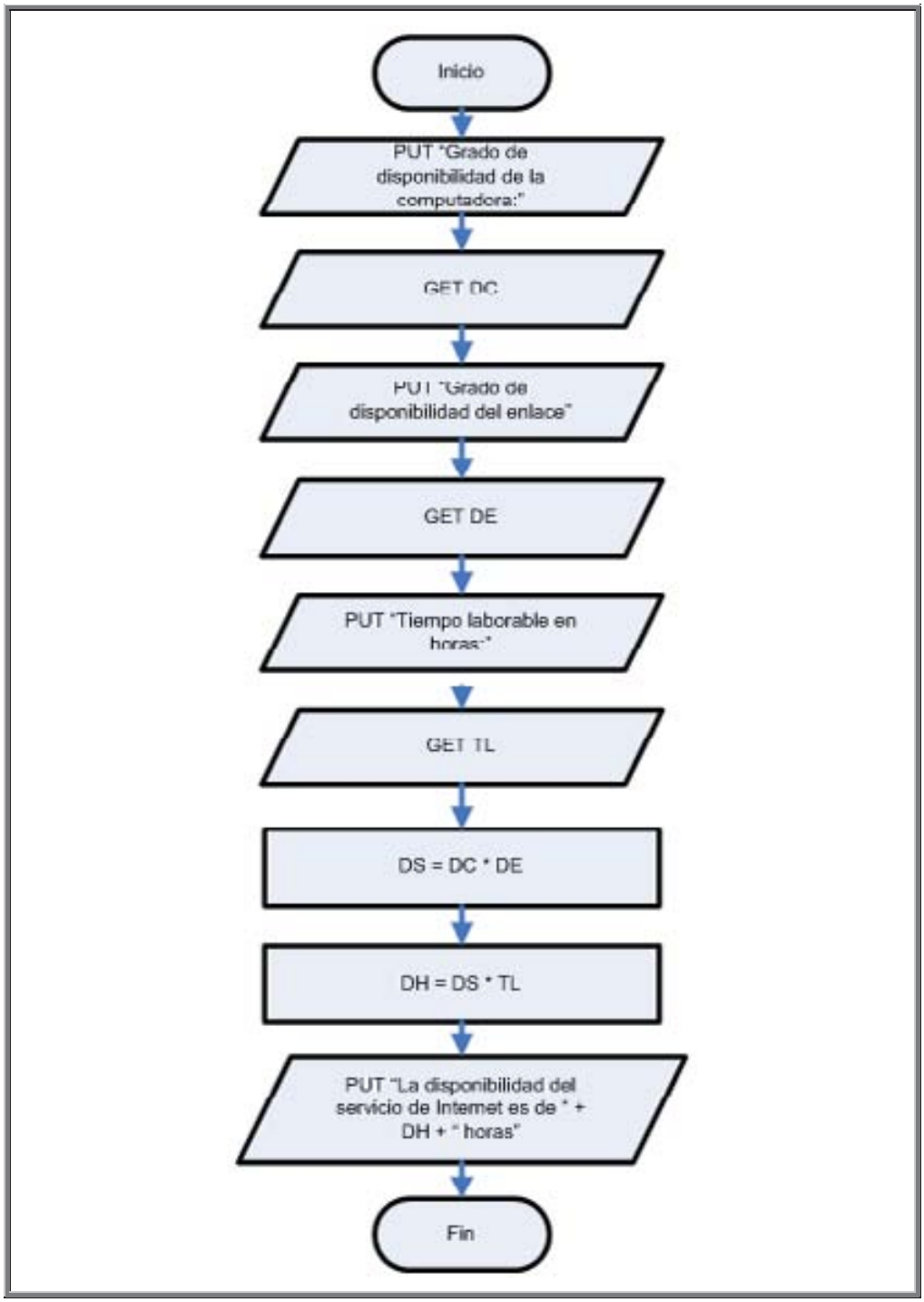
1. Inicio.
2. Se pregunta la disponibilidad de la computadora (DC,d[0-1]).
3. Se pregunta la disponibilidad del enlace (DE,d[0-1]).
4. Se pregunta el tiempo laborable en horas (TL,d[0-1]).
5. Se calcula la disponibilidad del sistema de dispositivos (DS,d[0-1]).
6. Se calcula el tiempo de disponibilidad del servicio (DH,d[0-1]).
7. Se muestra el tiempo de disponibilidad del servicio (DH)
8. Fin.

Miniespecificación

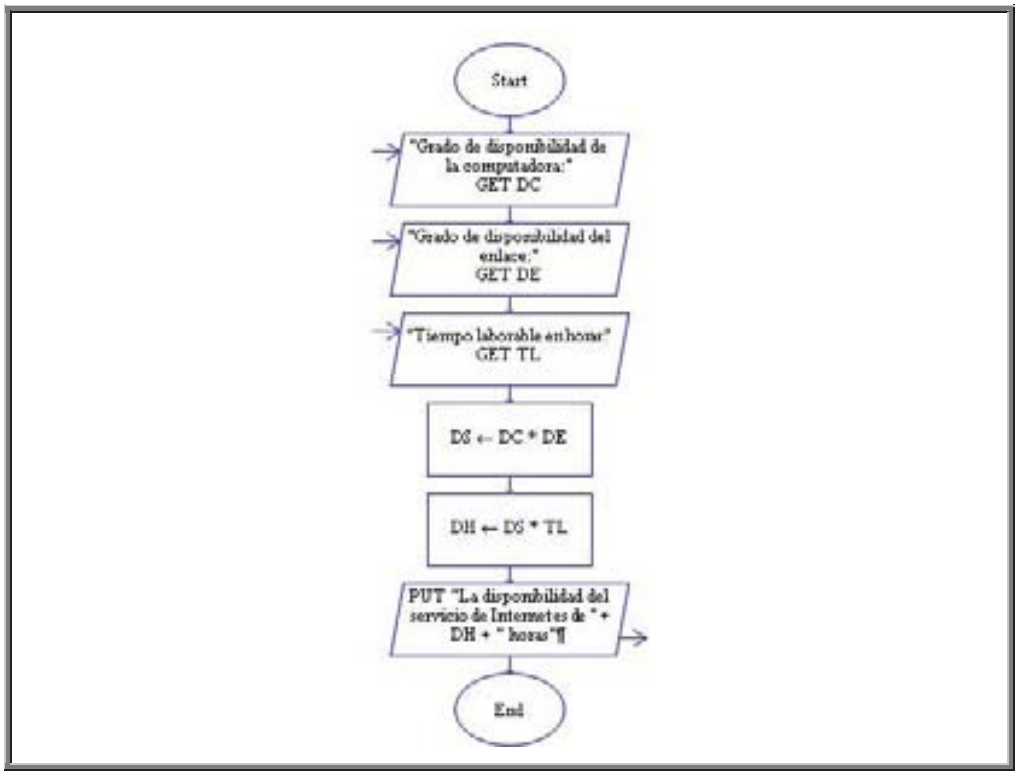
Dado que se trata de una petición y una muestra de datos, la selección de símbolos no podría ser más evidente.

```
1   INICIO:
2           (DC,d[0-1])
3           (DE,d[0-1])
4           (TL,d[0-1])
5           (DS,d[0-1])
6           (DH,d[0-1])
7           « "Grado de disponibilidad de la computadora:"
8           » DC
9           « "Grado de disponibilidad del enlace:"
10          » DE
11          « "Tiempo laborable en horas:"
12          » TL
13          DS ← DC * DE
14          DH ← DS * TL
15          « "La disponibilidad del servicio de Internet es de " + DH + " horas"
16  :FIN
```

Diagrama de flujo del programa



Programa en Raptor



Codificación de **C2Niveles.vb**

```
1   Imports System
2   Imports System.Console
3
4   Module Niveles
5       Sub Main()
6
7           Dim DC As Decimal
8           Dim DE As Decimal
9           Dim TL As Decimal
10          Dim DS As Decimal
11          Dim DH As Decimal
12
```

```
13          Write("Grado de disponibilidad de la computadora: ")
14          DC = ReadLine()
15          Write("Grado de disponibilidad del enlace: ")
16          DE = ReadLine()
17          Write("Tiempo laborable en horas: ")
18          TL = ReadLine()
19
20          DS= DC*DE
21          DH= Ds*TL
22
23          WriteLine("La disponibilidad del servicio de Internet es de " & DH &" horas")
24
25      End Sub
26  End Module
```

Codificación de **C2Niveles.cs**

```
1   using System;
2
3   class C2Niveles
4   {
5           static void Main()
6           {
7
8                   decimal DC;
9                   decimal DE;
10                  decimal TL;
11                  decimal DS;
12                  decimal DH;
13
14                  Console.Write("Grado de disponibilidad de la computadora: ");
15                  DC = decimal.Parse(Console.ReadLine());
16                  Console.Write("Grado de disponibilidad del enlace: ");
17                  DE = decimal.Parse(Console.ReadLine());
18                  Console.Write("Tiempo laborable en horas: ");
19                  TL = decimal.Parse(Console.ReadLine());
20
21                  DS= DC*DE;
22                  DH= DS*TL;
23
24                  Console.WriteLine("La disponibilidad del servicio de Internet es de "+
25                          DH +" horas");
26
27          }
28  }
```

Codificación de **C2Niveles.cpp**

```
1   #include <iostream>
2   using namespace std;
3
4   void main()
5   {
6
7           double DC;
8           double DE;
9           double TL;
10          double DS;
11          double DH;
12
13          cout << "Grado de disponibilidad de la computadora: ";
14          cin >> DC;
15          cout << "Grado de disponibilidad del enlace: ";
16          cin >> DE;
17          cout << "Tiempo laborable en horas:  ";
18          cin >> TL;
19
20          DS= DC*DE;
21          DH= DS*TL;
22
23          cout << "La disponibilidad del servicio de internet es de "<< DH << " horas" << endl;
24
25  }
```

Codificación de **C2Niveles.java**

```
1   import java.io.*;
2   import java.lang.*;
3
4   class C2Niveles {
5           public static void main(String args[])
6           throws IOException
7     {
8
9                   BufferedReader entrada = new
10                          BufferedReader(new InputStreamReader(System.in));
11
12                  float DC;
13                  float DE;
14                  float TL;
15                  float DS;
16                  float DH;
17
```

```
18                  System.out.print("Grado de disponibilidad de la computadora: ");
19                  DC=Float.parseFloat(entrada.readLine());
20                  System.out.print("Grado de disponibilidad del enlace: ");
21                  DE=Float.parseFloat(entrada.readLine());
22                  System.out.print("Tiempo laborable en horas:  ");
23                  TL=Float.parseFloat(entrada.readLine());
24
25                  DS= DC*DE;
26                  DH= DS*TL;
27
28                  System.out.println("La disponibilidad del servicio de internet es de "+
29                          DH +" horas");
30
31          }
32  }
```

Caso 3: Muestra de una tabla de multiplicar

El programa debe mostrar cualquier tabla de multiplicar que le solicitemos. La única restricción es que la tabla deberá ser de un número mayor o igual a 1, y menor o igual a 10.

Si se pide una tabla de un número que no esté entre el 1 y el 10, el programa debe indicar, mediante un mensaje, que no puede continuar.

Utilidad del programa:

1. Comprueba el uso de estatutos condicionales y estatutos de ciclo.
2. Comprueba el uso de los contadores.

Análisis del caso

Área del conocimiento: Matemática elemental.

Análisis de los elementos del todo:

1. Se debe mostrar una tabla de multiplicar.
2. La tabla a mostrar debe ser solicitada.
3. La tabla solicitada debe estar entre el uno y el 10.
4. Cualquier solicitud de tabla fuera del rango de 1 a 10 debe ser rechazada, provocando un mensaje de error.

Unidades de funcionalidad

ID	Unidad de funcionalidad	Prioridad	Obligatoriedad
1	Solicitar una tabla de multiplicar y desplegarla.	Alta	Si

Interfaz

Qué tabla deseas: _____

<Aquí se muestra el resultado>

Algoritmo

1. Inicio.
2. Se pregunta qué tabla se desea
3. Si la tabla está entre 1 y 10, entonces se imprime la tabla solicitada
4. Si la tabla no está entre el 1 y 10, entonces se muestra un mensaje de error.
5. Fin.

Simulación

Supongamos que nosotros le pedimos la tabla del 5, el programa debería mostrar lo siguiente.

```
Qué tabla deseas: 5
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
```

Supongamos que nosotros le pedimos la tabla del 25, el programa debería mostrar lo siguiente.

```
Qué tabla deseas: 25
Válido sólo del 1 al 10
```

Algoritmo orientado a los datos

1. Inicio.
2. Se pregunta qué tabla se desea (Tabla,i[1-10])
3. Si la tabla (Tabla) está entre 1 y 10, entonces se imprime la tabla solicitada
4. Si la tabla no está entre el 1 y 10, entonces se muestra un mensaje de error.
5. Fin.

Miniespecificación (sin utilizar estructuras de control)

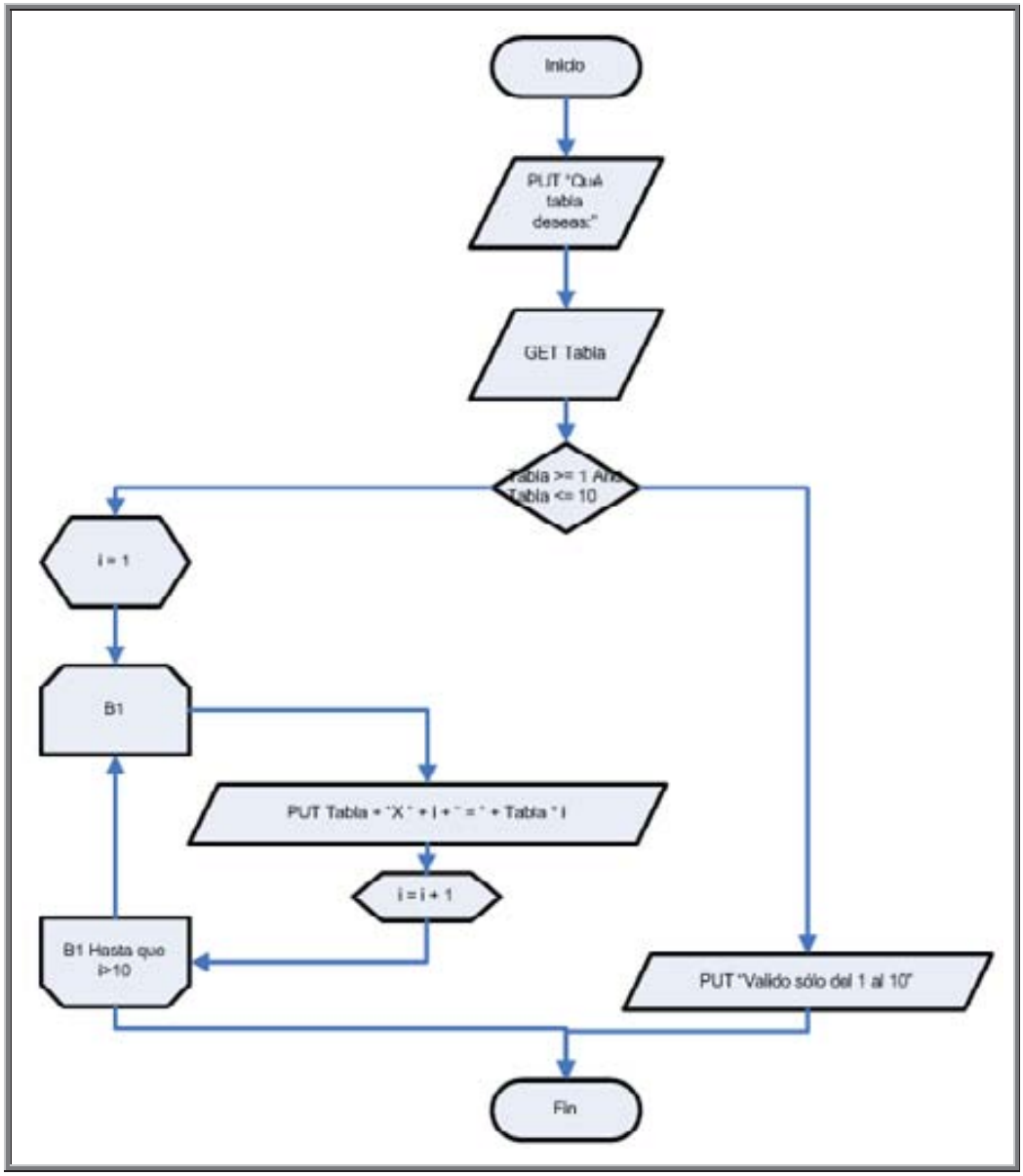
```
1   INICIO:
2           (Tabla,i[1-10])
3           « "Qué tabla deseas: "
4           » Tabla
5           Si Tabla >= 1 ∧ Tabla <=10 Entonces
6                   « Tabla + " X" + 1 + "=" + Tabla *1
7                   « Tabla + " X" + 2 + "=" + Tabla *2
8                   « Tabla + " X" + 3 + "=" + Tabla *3
9                   « Tabla + " X" + 4 + "=" + Tabla *4
10                  « Tabla + " X" + 5 + "=" + Tabla *5
11                  « Tabla + " X" + 6 + "=" + Tabla *6
12                  « Tabla + " X" + 7 + "=" + Tabla *7
13                  « Tabla + " X" + 8 + "=" + Tabla *8
14                  « Tabla + " X" + 9 + "=" + Tabla *9
15                  « Tabla + " X" + 10 + "=" + Tabla *10
16          De lo contrario
17                  « "Válido sólo del 1 al 10"
18          Fin Si
19  :FIN
```

Miniespecificación (usando estructuras de control)

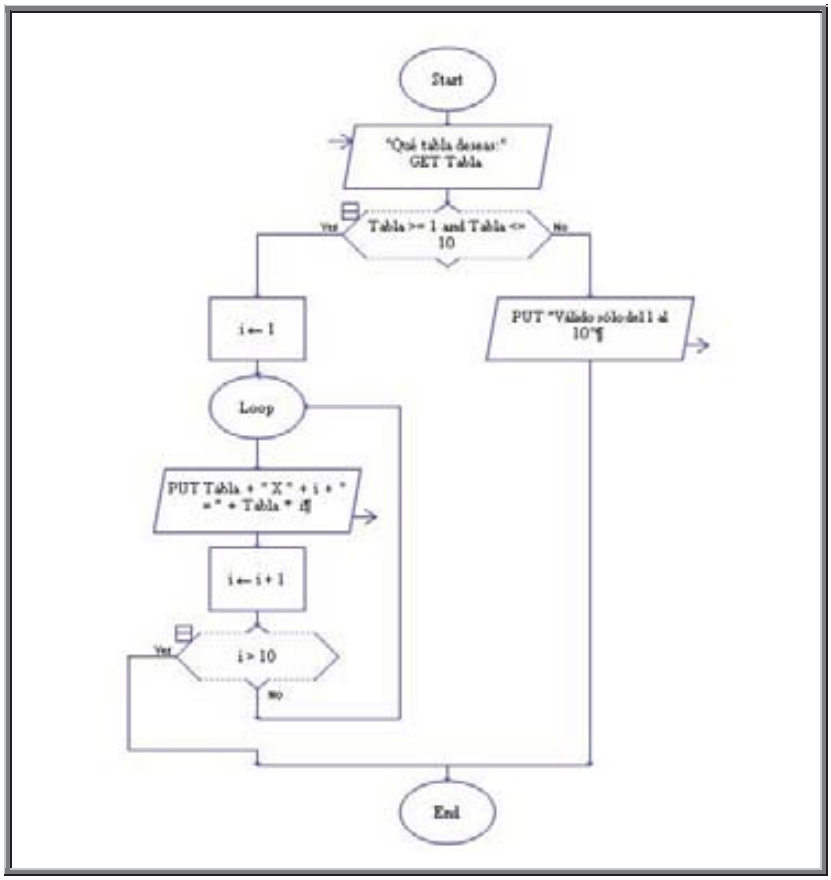
```
1   INICIO:
2           (i,i[0-n])
3           (Tabla,i[1-10])
4           « "Qué tabla deseas: "
5           » Tabla
6           Si Tabla >= 1 ∧ Tabla <=10 Entonces
7                   i ← 1
8                   Inicio
9                           « Tabla + " X" + i + "=" + Tabla * i
10                          i ← i + 1
11                  Hasta que i > 10
```

```
12          De lo contrario
13                  « "Válido sólo del 1 al 10"
14          Fin Si
15  :FIN
```

Diagrama de flujo del programa



Programa en Raptor



Codificación de **C3Tabla.vb**

```
1   Imports System
2   Imports System.Console
3
4   Module C3Tabla
5       Sub Main()
6
7           Dim Tabla As Integer
8           Dim i As Integer
9
10          Write("Qué tabla deseas: ")
11          Tabla = ReadLine()
```

```
12
13          If Tabla >=1 And Tabla <= 10 Then
14             i=1
15             Do
16                WriteLine(Tabla & " X " & i & " = "& Tabla*i )
17                i += 1
18             Loop Until i > 10
19          Else
20             WriteLine("Válido sólo del 1 al 10")
21          End If
22
23       End Sub
24    End Module
```

Codificación de **C3Tabla.cs**

```
1   using System;
2
3   class C3Tabla
4   {
5           static void Main()
6           {
7
8                   int Tabla;
9                   int i=1;
10
11                  Console.Write("Qué tabla deseas: ");
12                  Tabla =  int.Parse(Console.ReadLine());
13
14                  if(Tabla >= 1 & Tabla <=10)
15                  {
16                          i=1;
17                          do
18                           {
19                                  Console.WriteLine(Tabla + " X " + i + " = "+ Tabla * i );
20                                  i+=1;
21                           }
22                          while (i <=10);
23                  }
24                  else
25                  {
26                          Console.Write("Válido solo del 1 al 10");
27                  }
28
29          }
30  }
```

Codificación de **C3Tabla.cpp**

```
1   #include <iostream.h>
2   using namespace std;
3
4   int main()
5   {
6           int Tabla;
7           int i;
8
9           cout << "Qué tabla deseas: ";
10          cin >> Tabla;
11
12          if (Tabla >=1 & Tabla<=10 )
13          {
14                  i=1;
15                  do
16                  {
17                          cout << Tabla << " X "<< i << " = "<< Tabla*i << endl;
18                          i+=1;
19                  }
20                  while (i <= 10);
21          }
22          else
23          {
24                  cout << "Valido solo del 1 al 10" << endl;
25          }
26  }
```

Codificación de **C3Tabla.java**

```
1   import java.io.*;
2   import java.lang.*;
3
4   class C3Tabla
5   {
6           public static void main(String args[])
7           throws java.io.IOException
8     {
9
10                  BufferedReader entrada = new
11                          BufferedReader(new InputStreamReader(System.in));
12                  int i=0;
13                  int Tabla;
14
15                  System.out.print("Qué tabla deseas: ");
16                  Tabla=Integer.parseInt(entrada.readLine());
```

```
17
18                  if (Tabla >=1 & Tabla<=10 )
19                  {
20
21                          do
22                          {
23
24                                  System.out.println(Tabla + " X " + i + " = " + Tabla*i );
25                                  i+=1;
26                          }
27                          while (i <= 10);
28                  }
29                  else
30                  {
32                          System.out.println("Valido solo del 1 al 10");
33                  }
34
35          }
36  }
```

Caso 4: Muestra las tablas de multiplicar del 1 al 5

El programa debe mostrar todas las tablas de multiplicar del 1 al 5, incluyendo un encabezado por tabla.

Utilidad del programa:

1. Comprueba el uso de estructuras anidadas.

Análisis del caso

Área del conocimiento: Matemática elemental.

Análisis de los elementos del todo:

1. Se deben mostrar las tablas de multiplicar del 1, 2, 3, 4 y 5.

Unidades de funcionalidad

ID	Unidad de funcionalidad	Prioridad	Obligatoriedad
1	Desplegar las tablas de multiplicar del 1 al 5.	Alta	Si

Interfaz

<Muestra las tablas del 1 al 5 aquí>

Algoritmo

1. Inicio.
2. Muestra el encabezado de la tabla del 1
3. Se calcula la tabla del 1 y se muestra
4. Muestra el encabezado de la tabla del 2
5. Se calcula la tabla del 2 y se muestra
6. Muestra el encabezado de la tabla del 3
7. Se calcula la tabla del 3 y se muestra
8. Muestra el encabezado de la tabla del 4
9. Se calcula la tabla del 4 y se muestra
10. Muestra el encabezado de la tabla del 5
11. Se calcula la tabla del 5 y se muestra
12. Fin.

Simulación

No se proporcionan datos. El programa es de sólo salida, y muestra lo siguiente.

```
TABLA: 1
1 X 1 = 1
1 X 2 = 2
1 X 3 = 3
1 X 4 = 4
1 X 5 = 5
1 X 6 = 6
1 X 7 = 7
1 X 8 = 8
1 X 9 = 9
1 X 10 = 10
TABLA: 2
2 X 1 = 2
2 X 2 = 4
...
...
...
5 X 6 = 30
5 X 7 = 35
5 X 8 = 40
5 X 9 = 45
5 X 10 = 50
```

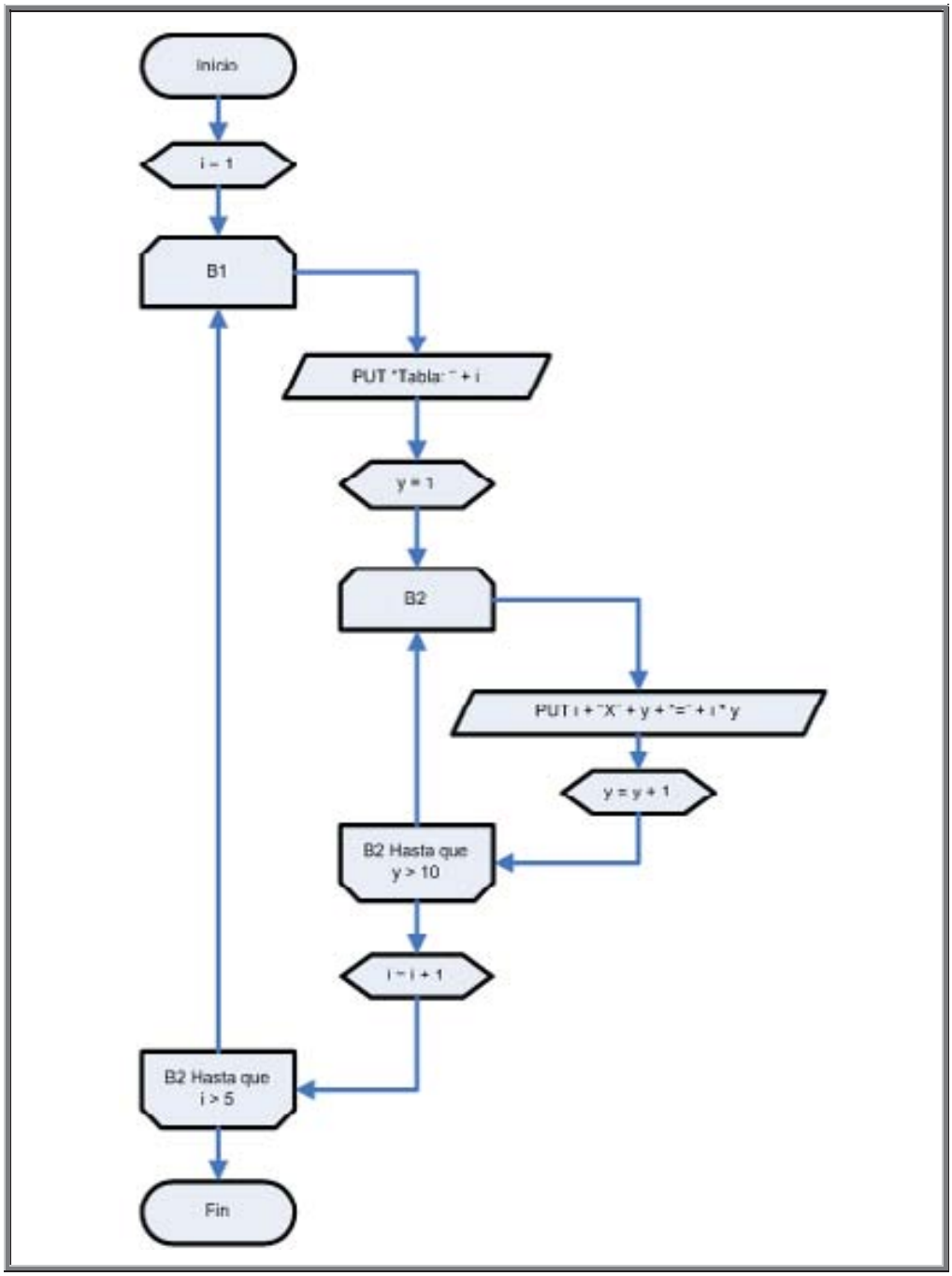
Algoritmo orientado a datos (igual a Algoritmo)

1. Inicio.
2. Muestra el encabezado de la tabla del 1
3. Se calcula la tabla del 1 y se muestra
4. Muestra el encabezado de la tabla del 2
5. Se calcula la tabla del 2 y se muestra
6. Muestra el encabezado de la tabla del 3
7. Se calcula la tabla del 3 y se muestra
8. Muestra el encabezado de la tabla del 4
9. Se calcula la tabla del 4 y se muestra
10. Muestra el encabezado de la tabla del 5
11. Se calcula la tabla del 5 y se muestra
12. Fin.

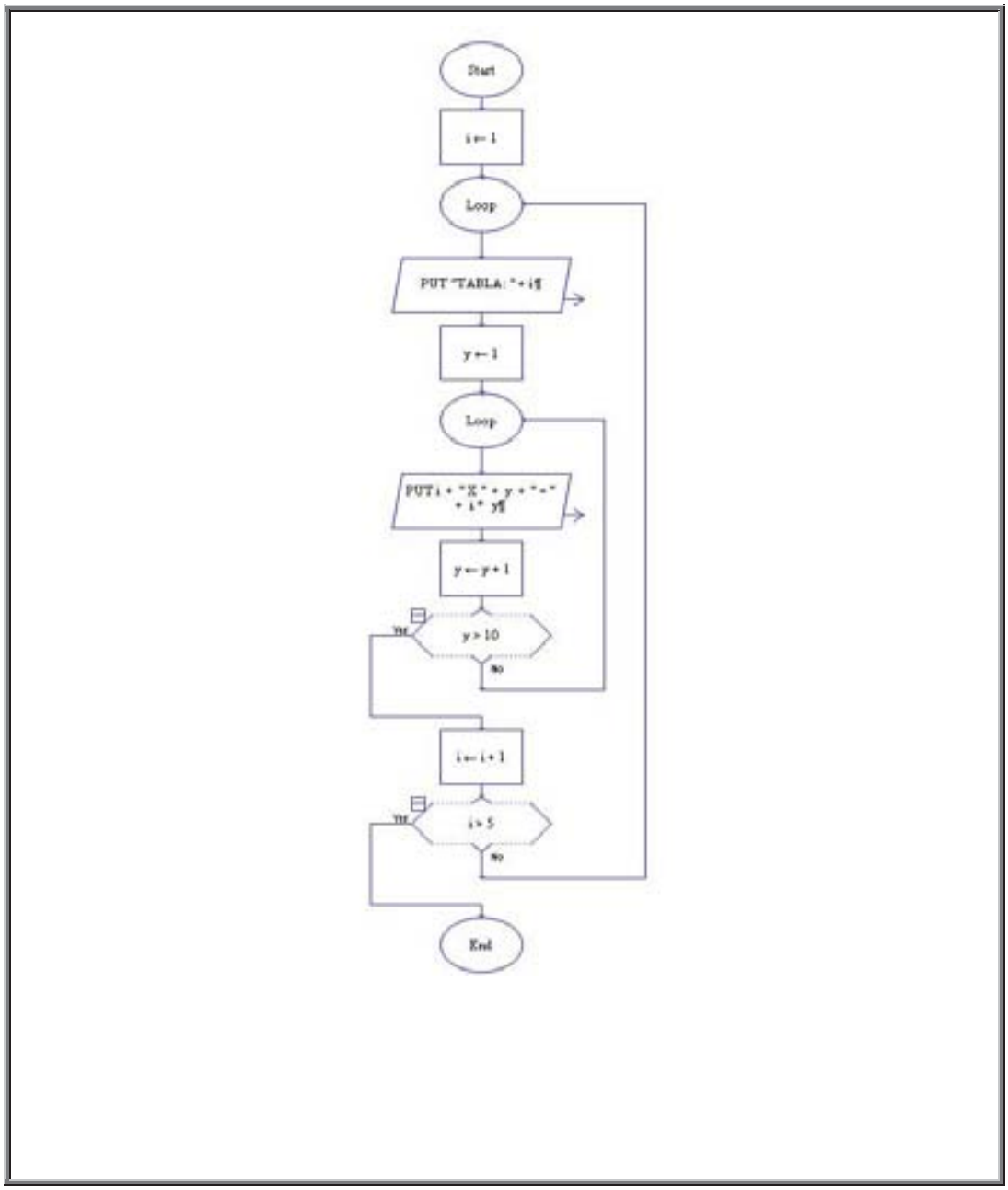
Miniespecificación

```
1   INICIO:
2           (i,i[0-n])
3           (y,i[0-n])
4           i ← 1
5           Inicio
6                   « "TABLA: " + i
7                   y ← 1
8                   Inicio
9                           « i + " X" + y + "=" + i * y
10                          y ← y + 1
11                  Hasta que y > 10
12                  i ← i + 1
13          Hasta que i > 5
14  :FIN
```

Diagrama de flujo del programa



Programa en Raptor



Codificación de **C4MultiTabla.vb**

```
1   Imports System
2   Imports System.Console
3
4   Module C4MultiTabla
5       Sub Main()
6
7           Dim i As Integer
8           Dim y As Integer
9
10          i=1
11          Do
12              WriteLine("Tabla: " & i)
13              y=1
14              Do
15                  WriteLine(i & " X " & y & " = "& i*y )
16                  y += 1
17              Loop Until y > 10
18              i += 1
19          Loop Until i > 5
20
21      End Sub
22  End Module
```

Codificación de **C4MultiTabla.cs**

```
1   using System;
2
3   class C4MultiTabla
4   {
5           static void Main()
6           {
7
8                   int i;
9                   int y;
10
11                  i = 1;
12                  do
13                  {
14                          Console.WriteLine("Tabla: "+i);
```

```
15                              y = 1;
16                              do
17                              {
18                                      Console.WriteLine(i + " X " + y + " = "+ i*y );
19                                  y+=1;
20                              }
21                              while (y <=10 );
22                              i+=1;
23                      }
24                      while (i <= 5);
25
26              }
27  }
```

Codificación de **C4MultiTabla.cpp**

```
1   #include <iostream.h>
2   using namespace std;
3
4   int main()
5   {
6
7       int i;
8               int y;
9
10              i=1;
11              do
12              {
13                      cout << "Tabla: "<< i << endl;
14                      y=1;
15                      do
16                      {
17                              cout << i << " X "<< y << " = "<< i*y << endl;
18                              y += 1;
19                      }
20                      while (y <= 10);
21                      i += 1;
22              }
23              while (i <= 5);
24  }
```

Codificación de **C4MultiTabla.java**

```
1   import java.io.*;
2   import java.lang.*;
3
4   class C4MultiTabla
5   {
6           public static void main(String args[])
7           throws java.io.IOException
8     {
9
10                  int i=1;
11                  int y=1;
12
13                  i=1;
14                  do
15                  {
16                          System.out.println("Tabla: " + i );
17                          y=1;
18                          do
19                          {
20                                  System.out.println(i + " X " + y + " = " + i*y );
21                                  y += 1;
22                           }
23                          while (y <= 10);
24                          i += 1;
25                  }
26                  while (i <= 5);
27
28    }
29  }
```

Caso 5: Análisis de promedios

El programa debe recopilar 5 números, para después mostrarlos indicando si están o no por encima de su promedio. Al final deberá enviarse un resumen en donde se indique la suma total de los números, y el promedio calculado.

Para la recopilación de datos utilice un arreglo.

Utilidad del programa:

1. Comprueba el uso de acumuladores.
2. Comprueba el uso de los arreglos.

Análisis del caso

Área del conocimiento: Matemática elemental.

Análisis de los elementos del todo:

1. El programa recopila 5 números.
2. El programa suma los números proporcionados.
3. El programa calcula el promedio de los números proporcionados.
4. El programa determina si cada uno de los números está por encima o no de su promedio.
5. El programa muestra la suma total calculada.
6. El programa muestra la comparación de cada uno de los números respecto al promedio calculado.

Unidades de funcionalidad

ID	Unidad de funcionalidad	Prioridad	Obligatoriedad
1	Recopilar una serie de números, calcular el promedio de la serie, y mostrar cada uno de los números proporcionados, indicando si están o no por encima del promedio.	Alta	Si

Interfaz

Valor para el elemento 1: _____
Valor para el elemento 2: _____
Valor para el elemento 3: _____

```
Valor para el elemento 4: _________
Valor para el elemento 5: _________
<Resultado del programa>
```

Algoritmo

1. Inicio.
2. Se pregunta el valor del elemento 1
3. Se pregunta el valor del elemento 2
4. Se pregunta el valor del elemento 3
5. Se pregunta el valor del elemento 4
6. Se pregunta el valor del elemento 5
7. Se realiza la suma de todos los elementos
8. Se calcula el promedio de la serie de números
9. Se muestra la lista de valores, informando si el número se encuentra por encima o no del promedio
10. Se muestra el resultado de la suma
11. Se muestra el resultado del promedio
12. Fin.

Simulación

Supongamos que nosotros le proporcionamos los siguientes valores al programa: 10, 15, 25, 9, 4. La suma de los valores da 63. Dividiendo la suma entre 5 (número de elementos), se tiene un promedio de 12.6.

```
Valor para el elemento 1: 10
Valor para el elemento 2: 15
Valor para el elemento 3: 25
Valor para el elemento 4: 9
Valor para el elemento 5: 4

Número 10 menor al promedio
Número 15 mayor o igual al promedio
Número 25 mayor o igual al promedio
Número 9 menor al promedio
Número 4 menor al promedio
Suma: 63
Promedio: 12.6000
```

Algoritmo

1. Inicio.
2. Se pregunta el valor del elemento 1 (Entrada(1),i[0-n])
3. Se pregunta el valor del elemento 2 (Entrada(2),i[0-n])
4. Se pregunta el valor del elemento 3 (Entrada(3),i[0-n])

5. Se pregunta el valor del elemento 4 (Entrada(4),i[0-n])
6. Se pregunta el valor del elemento 5 (Entrada(5),i[0-n])
7. Se realiza la suma de todos los elementos (Suma,d[0-n])
8. Se calcula el promedio de la serie de números (Promedio,d[0-n])
9. Se muestra la lista de valores (Entrada), informando si el número (Entrada(n))se encuentra por encima o no del promedio (Promedio)
10. Se muestra el resultado de la suma (Suma)
11. Se muestra el resultado del promedio (Promedio)
12. Fin.

Miniespecificación (sin usar estructuras)

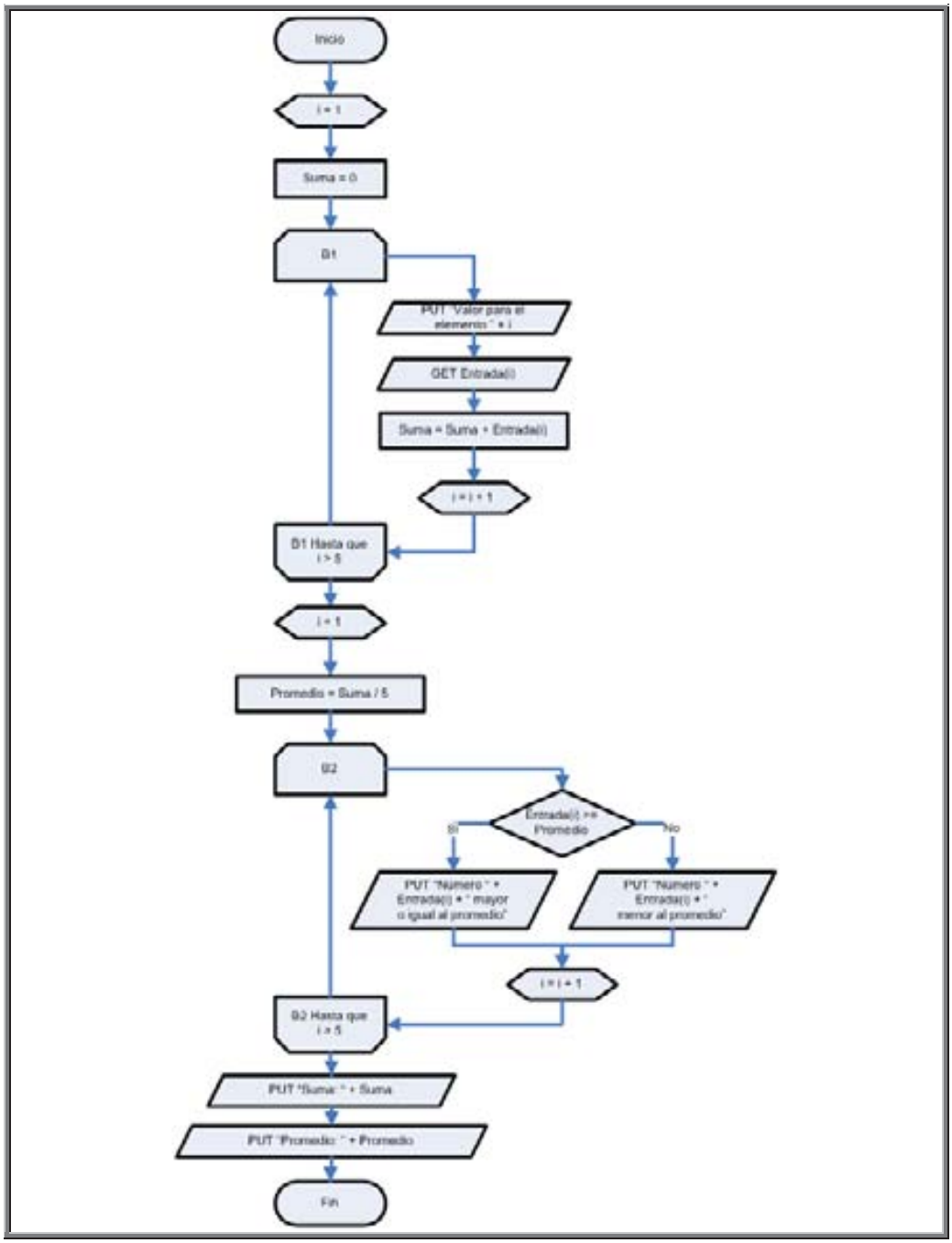
```
1   INICIO:
2           (Entrada(5),i[0-n])
3           (Suma,d[0-n])
4           (Promedio,d[0-n])
5           Suma ← 0
6           « "Valor para el elemento 1:"
7           » Entrada(1)
8           « "Valor para el elemento 2:"
9           » Entrada(2)
10          « "Valor para el elemento 3:"
11          » Entrada(3)
12          « "Valor para el elemento 4:"
13          » Entrada(4)
14          « "Valor para el elemento 5:"
15          » Entrada(5)
16          Suma ← Entrada(1) + Entrada(2) + Entrada(3) + Entrada(4) + Entrada(5)
17          Promedio ← Suma / 5
18          Si Entrada(1) >= Promedio Entonces
19              « "Número " + Entrada(1) + " mayor o igual al promedio: "
20          De lo contrario
21              « "Número " + Entrada(1) + " menor al promedio: "
22          Fin Si
23          Si Entrada(2) >= Promedio Entonces
24              « "Número " + Entrada(2) + " mayor o igual al promedio: "
25          De lo contrario
26              « "Número " + Entrada(2) + " menor al promedio: "
27          Fin Si
28          Si Entrada(3) >= Promedio Entonces
29              « "Número " + Entrada(3) + " mayor o igual al promedio: "
30          De lo contrario
31              « "Número " + Entrada(3) + " menor al promedio: "
32          Fin Si
33          Si Entrada(4) >= Promedio Entonces
34              « "Número " + Entrada(4) + " mayor o igual al promedio: "
35          De lo contrario
36              « "Número " + Entrada(4) + " menor al promedio: "
37          Fin Si
38          Si Entrada(5) >= Promedio Entonces
39
```

```
40                  « "Número " + Entrada(5) + " mayor o igual al promedio: "
41              De lo contrario
42                  « "Número " + Entrada(5) + " menor al promedio: "
43              Fin Si
44              « "Suma: " + Suma
45              « "Promedio: " + Promedio
    :FIN
```

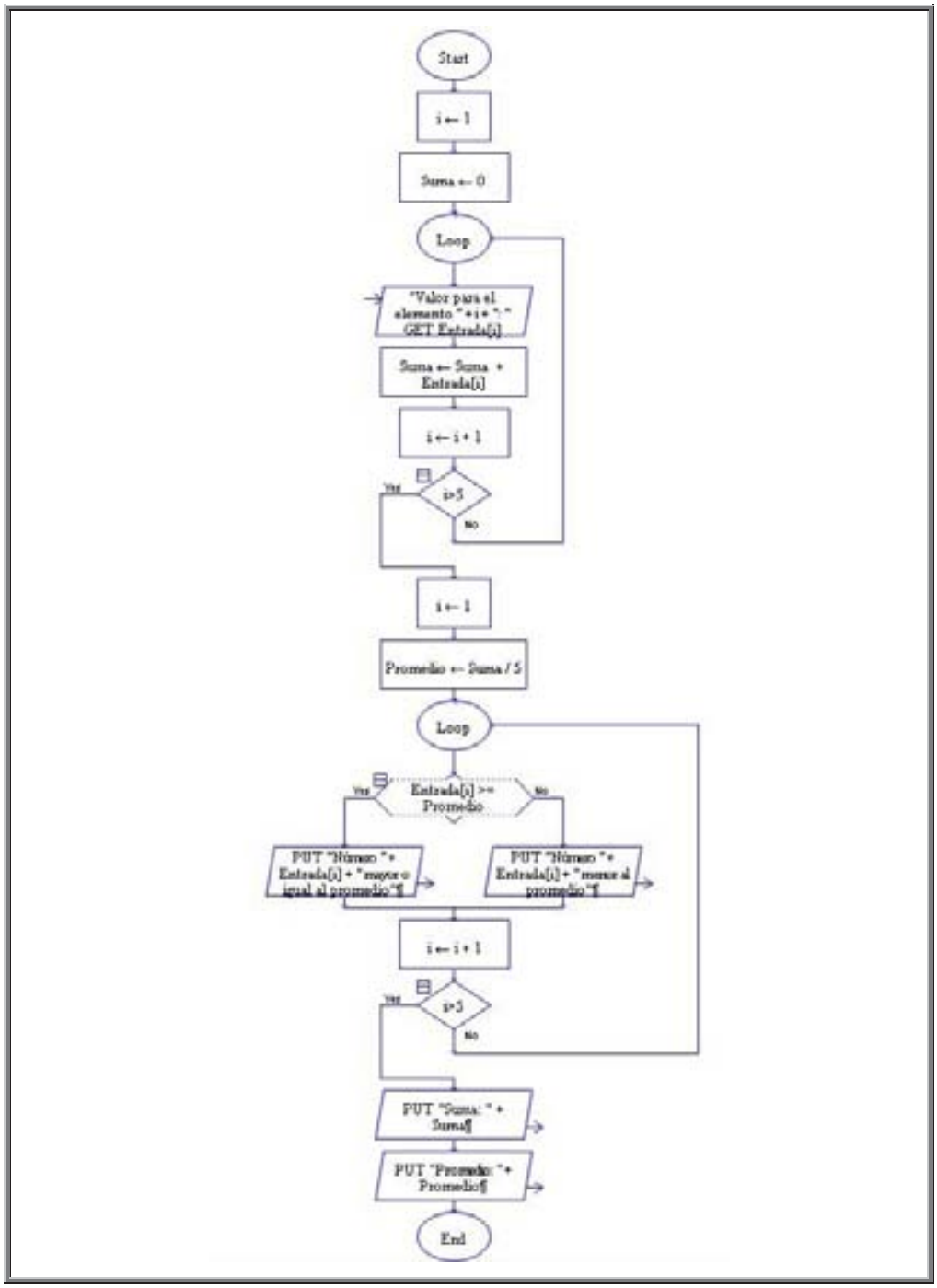
Miniespecificación (usando estructuras)

```
1   INICIO:
2               (i,i[0-n])
3               (Entrada(5),i[0-n])
4               (Suma,d[0-n])
5               (Promedio,d[0-n])
6               Suma ← 0
7               i ← 1
8               Inicio
9                           « "Valor para el elemento " + i + ": "
10                          » Entrada(i)
11                          Suma ← Suma + Entrada(i)
12                          i ← i + 1
13              Hasta que i > 5
14              Promedio ← Suma / 5
15              i ← 1
16              Inicio
17                          Si Entrada(i) >= Promedio Entonces
18                              « "Número " + Entrada(i) + " mayor o igual al promedio. "
19                          De lo contrario
20                              « "Número " + Entrada(i) + " menor al promedio. "
21                          Fin Si
22                          i ← i + 1
23              Hasta que i > 5
24              « "Suma: " + Suma
25              « "Promedio: " + Promedio
26  :FIN
```

Diagrama de flujo del programa



Programa en Raptor



Codificación de **C5Promedios.vb**

```
1   Imports System
2   Imports System.Console
3
4   Module C5Promedios
5      Sub Main()
6
7         Dim i as Integer
8         Dim Entrada(4)As Integer
9         Dim Suma as Integer
10        Dim Promedio As Decimal
11        Dim Valor as Integer
12
13        Suma=0
14
15        i=1
16        Do
17           Write("Valor para el elemento " & i & ": ")
18           Entrada(i-1)=ReadLine()
19           Suma+=Entrada(i-1)
20           i += 1
21        Loop Until i > 5
22
23        Promedio= Suma/5
24
25        i=1
26        Do
27           If Entrada(i-1)>= Promedio Then
28              WriteLine("Número " & Entrada(i-1) & " mayor o igual al promedio.")
29           Else
30              WriteLine("Número " & Entrada(i-1) & " menor al promedio.")
31           End If
32           i += 1
33        Loop Until i > 5
34
35        WriteLine("Suma: " & Suma)
36        WriteLine("Promedio: " & Promedio )
37
38     End Sub
39  End Module
```

Codificación de **C5Promedios.cs**

```
1   using System ;
2
3   class C5Promedios
4   {
5           static void Main ( string[ ] args )
6           {
7
8                   int i;
9                   int[] Entrada = new int[5];
10                  int Suma;
11                  decimal Promedio;
12
13                  Suma=0;
14
15                  i=1;
16                  do
17                  {
18                          Console.Write("Valor para el elmento " + i + ": ");
19                          Entrada[i-1] = int.Parse(Console.ReadLine());
20                          Suma+=Entrada[i-1];
21                          i+=1;
22                  }
23                  while (i <=5 );
24
25                  Promedio=Suma/5;
26
27                  i=1;
28                  do
29                  {
30                          if(Entrada[i-1] >= Promedio)
31                          {
32                                  Console.WriteLine("Número " + Entrada[i-1] +
33                                          " mayor o igual al promedio");
34                          }
35                          else
36                          {
37                                  Console.WriteLine("Número " + Entrada[i-1] +
38                                          " menor al promedio");
39                          }
40                          i+=1;
41                  }
42                  while (i <=5 );
43
44                  Console.WriteLine("Suma: " + Suma );
45                  Console.WriteLine("Promedio: " + Promedio );
46
47          }
48  }
```

Codificación de **C5Promedios.cpp**

```
1   #include <iostream.h>
2   using namespace std;
3
4   int main()
5   {
6
7           int i;
8           int Entrada[5];
9           int Suma;
10          float Promedio;
11
12          Suma=0;
13
14          i=1;
15          do
16          {
17                  cout << "Valor para el elmento " << i << ": " ;
18                  cin >> Entrada[i-1];
19                  Suma+=Entrada[i-1];
20                  i += 1;
21          }
22          while (i <= 5);
23
24          Promedio = Suma / 5;
25
26          i=1;
27          do
28          {
29                  if (Entrada[i-1] >= Promedio)
30                  {
31                    cout << "Numero " << Entrada[i-1] << " mayor igual al promedio" << endl;
32                  }
33                  else
34                  {
35                    cout << "Numero " << Entrada[i-1] << " menor al promedio" << endl;
36                  }
37                  i += 1;
38          }
39          while (i <= 5);
40
41          cout << "Suma: " << Suma << endl;
42          cout << "Promedio: " << Promedio << endl;
43
44  }
45
46
```

Codificación de **C5Promedios.java**

```
1   import java.io.*;
2   import java.lang.*;
3
4   class C5Promedios
5   {
6           public static void main(String args[])
7           throws IOException
8     {
9                   int i;
10                  int[] Entrada = new int[5];
11                  int Suma;
12                  float Promedio;
13                  BufferedReader entrada = new
14                          BufferedReader(new InputStreamReader(System.in));
15                  Suma = 0;
16                  i=1;
17                  do
18                  {
19                          System.out.print("Valor para el elmento "+  i + ": ") ;
20                          Entrada[i-1] = Integer.parseInt(entrada.readLine());
21                          Suma += Entrada[i-1];
22                          i += 1;
23                  }
24                  while (i <= 5);
25                  Promedio=Suma/5;
26                  i=1;
27                  do
28                  {
29                          if (Entrada[i-1] >= Promedio)
30                          {
31                                  System.out.println("Numero " + Entrada[i-1] +
32                                          " mayor igual al promedio.");
33                          }
34                          else
35                          {
36                                  System.out.println("Numero " + Entrada[i-1] +
37                                          " menor al promedio.");
38                          }
39                          i += 1;
40                  }
41                  while (i <= 5);
42
43                  System.out.println("Suma: " + Suma );
44                  System.out.println("Promedio: " + Promedio );
45          }
46  }
```

Anexo 2:

Tablas de referencia

Contenido

Tipos de datos base en VB.NET, C#, C++ y Java 476
Comparativo de operadores VB.NET, C#, C++ y Java 476
Símbolos ANSI/ISO 5807-1985 y Raptor 477
Comparativo de comandos VB.NET, C#, C++ y Java 478
Generalidades de VB.NET, C#, C++ y Java 482

En el presente anexo se muestran tablas de referencia utilizables en la resolución completa de los casos prácticos del libro.

www.detodoprogramacion.com

Tipos de datos base en VB.NET, C#, C++ y Java

Tipo común	VB.NET	C#	C++	Java
Booleano (l)	Boolean	bool	bool	boolean
Byte (b)	Byte	byte	short	byte
Cadena (x)	String	String	string	String
Decimal (d)	Decimal	decimal	double	long
Entero (i)	Integer	int	int	int

Comparativo de operadores VB.NET, C#, C++ y Java

Nombre	Visual Basic.NET	C#	C++	Java	Lógica simbólica
Igual	=	=	=	=	←
Concatenación	&	+	+	+	+
Suma	+	+	+	+	+
Resta	-	-	-	-	-
Multiplicación	*	*	*	*	*
División	/	/	/	/	/
Exponenciación	^	^	^	^	^
Disyuntiva	And	&&	&&	&&	∧
Conjuntiva	Or	||	||	||	∨
Negación	Not	!	!	!	¬
Mayor que	>	>	>	>	>
Menor que	<	<	<	<	<
Mayor o igual que	>=	>=	>=	>=	>=
Menor o igual que	<=	<=	<=	<=	<=
Igual a	=	==	==	==	=
Distinto a	<>	<>	<>	<>	<>
Suma incluyente	+=	+=	+=	+=	x ← x+1
Resta incluyente	-=	-=	-=	-=	x ← x-1

Símbolos ANSI/ISO 5807-1985 y Raptor

Símbolo	Equivalencia Raptor	Nombre
		Símbolo Datos
		Símbolo Proceso
		Símbolo Proceso predefinido
		Símbolo Preparación
		Símbolo Decisión
		Símbolo Modo paralelo
		Símbolo Límites de bucle
		Símbolo Línea de flujo
		Símbolo Línea punteada
		Símbolo Conector
	Start End	**Símbolo Terminador**
		Símbolo Anotación

Comparativo de comandos VB.NET, C#, C++ y Java

Asignación (Miniespecificación)	
Variable ← Valor	
Visual Basic.NET	C#
x = Valor	x = Valor;
C++	Java
x = Valor;	x = Valor;

Declaración de variables (Miniespecificación)	
(Variable,tipo[Dominio])	
Visual Basic.NET	C#
Dim x As Tipo	tipo x;
C++	Java
tipo x;	tipo x;

Declaración de arreglos (Miniespecificación)	
(Variable(elementos),tipo[Dominio])	
Visual Basic.NET	C#
Dim x(SubindiceMáximo) As Tipo	tipo[] x = new tipo[Elementos];
C++	Java
tipo x[Elementos];	tipo[] x = new tipo[Elementos];

Mostrado de datos (Miniespecificación)	
« Expresión	
Visual Basic.NET	C#
Console.WriteLine(Expresión)	Console.WriteLine(Expresión);
C++	Java
cout << Expresión;	System.out.println(Expresión);

Petición de datos (Miniespecificación)	
» Variable	
Visual Basic.NET	C#
x = Console.ReadLine()	x = Console.WriteReadLine();
C++	Java
cin >> x;	entrada.readLine();

Estructura condicional (Miniespecificación)

```
Si Condición Entonces
        Instrucciones
De lo contrario
        Instrucciones
Fin Si
```

Visual Basic.NET	C#

If ExpresiónLógica Then
  Instrucciones
Else
  Instrucciones
End If
	if (ExpresiónLógica)
	{
	  Instrucciones
	}
	else
	{
	  Instrucciones
	}

C++	**Java**

if (ExpresiónLógica)
{
  Instrucciones
}
else
{
  Instrucciones
}
	if (ExpresiónLógica)
	{
	  Instrucciones
	}
	else
	{
	  Instrucciones
	}

Estructura de control Incremental (Miniespecificación)

```
i ← ValorInicio
Inicio
        Instrucciones
        i ← i + 1;
Hasta que Condición
```

Visual Basic.NET	C#

i = ValorInicio
Do
Instrucciones
i += 1
Loop Until (i > ValorFin)
	i = ValorInicio;
	do
	{
	Instrucciones
	i += 1;
	}
	while (i <= ValorFin);

C++	**Java**

i = ValorInicio;
do
{
Instrucciones
i += 1;
}
while (i <= ValorFin);
	i = ValorInicio;
	do
	{
	Instrucciones
	i += 1;
	}
	while (i <= ValorFin);

Generalidades de VB.NET, C#, C++ y Java

Nombre	Visual Basic.NET	C#	C++	Java
Extensión del código fuente	.vb	.cs	.cpp	.java
Compilador	vbc.exe	csc.exe	bcc32.exe	javac.exe
Marco requerido de ejecución	.NET Framework	.NET Framework	N/A	J2SE
Herramienta base de desarrollo	.NET Framework SDK	.NET Framework SDK	Borland C++ Builder	JDK
La compilación genera	assembly	assembly	código máquina	bytecode
Extensión de archivo compilado	.exe	.exe	.exe	.class
Compila a código intermedio	Si	Si	No	Si
Nombre de biblioteca de clases	BCL	BCL	N/A	JFC
Clases almacenadas en archivos	.dll	.dll	.h	.h
Tiene jerarquía de clases	Si	Si	Si	Si
Referencia lógica de jerarquía usando	Imports	using	using	import

Índice temático

.class, 369
.H, 337
.OBJ, 337
« , 143
» , 141

a

Academia de la Fuerza Aérea de los Estados Unidos, 210
Acumuladores, 146
Alcance de un programa, 106
Algoritmo de programación, 109
Algoritmo, 109
Algoritmos orientados a datos, 138
Ambiente de desarrollo, 269
American National Standard Institute, 178
Amplitud de dominio, 65
Análisis, 102
Análisis, 25
Analizador de código, 25
Anidamiento, 149
Annotation, 180
ANSI, 178
Area de resultados, 216
Area de seguimiento, 214
Area de símbolos, 214
Area de trabajo, 214
Argumento, 37
Argumentos implícitos, 409
Arreglos, 150
Arreglos, 60
Assembly, 282
Assignment, 218

b

Barra de herramientas, 213
Barra de menú, 213
Barra de menú, 215
Base class, 405
Base Class Library, 271
BCC32.EXE, 337
BCL, 271
Biblioteca de clases base, 271
Biblioteca de clases, 270
Binary Large Object, 58
Bit, 6
BLOB, 58
Bloque de código, 20
Borland C++ Builder, 337
Borland C++ Compiler 5.5, 337
Borland Developer Studio, 337
Borland Jbuilder, 369
Botón de limpiado, 216
Breakpoints, 244
Bucle de comparación al final, 148
Bucle de comparación al inicio, 146
Bucle, 146
Buffer, 375
BufferReader, 375
Byte, 6
Bytecode, 369

www.detodoprogramacion.com

c

Cadenas, 20
Call, 218
Características de Raptor, 212
Case-sensitive, 305
Central Processing Unit, 4
Ciclo de desarrollo, 95
Clase, 403
Clase base, 405
Clase derivada, 405
Clase pública, 419
Clear, 216
Cliente, 103
Código controlado, 143
Código fuente, 23
Código intermedio de .NET, 282
Código intermedio de Java, 381
Código intermedio, 24
Código interpretado, 24
Código máquina, 24
Código objeto, 24
Command line, 216
Compilador de programas, 270
Compiladores de dos fases, 25
Compiladores de una sola fase, 25
Compiladores, 25
Comportamiento, 409
Computadora, 3
Conclusión, 38
Condición de selección, 240
Condicional compuesta, 144
Condicional simple, 144
Condicional, 144
Conjunto universo, 66
Connector, 180
Conocimiento, 8
Consola maestra, 213
Constructor, 403
Contadores, 145
CPU, 4
csc.exe, 303

d

Dashed line, 180
Data, 179
Dato de entrada, 42
Dato, 8
Datos de salida, 42
Decision, 179
Declaración, 140
Delimitador de fin, 143
Delimitador de inicio, 143
Delimitadores de la estructura, 143
Derived class, 405
Destructor, 403
Dev-C++, 337
Diagrama de flujo de programa, 178
Diagrama de flujo, 177
Diagrama de flujo, 178
Diagrama, 176
Direcciones de memoria, 4
Dispositivos de almacenamien-to permanente, 4
Dispositivos de entrada, 4
Dispositivos de procesamiento, 4
Dispositivos de salida, 4
Dispositivos periféricos, 4
Divulgación de datos, 11
Dominio de regla de negocio, 65
Dominio de relación, 65
Dominio de tipo, 65
Dominio, 58

e

Editor de programas, 269
Elemento, 150
Elemento, 61
Encapsulamiento, 404
Enlazador, 337
Ensamblado, 282
Ensamblados, 271
Entorno operativo, 5
Enunciado, 37
Enunciados declarativos, 38
Enunciados dependientes, 46
Espacio de nombres, 271
Estado, 140
Estatutos auxiliares de herencia, 419
Estructura de control, 146
Estructura de decisión, 144
Estructura del código, 20

Evento, 29, 404
Exclusiones, 67
Expresión absoluta, 59
Expresión compuesta, 59
Expresión lógica compuesta, 80
Expresión lógica simple, 80
Expresión, 59
Expresiones, 20

f

Fase de análisis, 25
Fase de análisis, 95
Fase de codificación, 96
Fase de diseño, 96
Fase de pruebas e implementación, 96
Finalidad del enunciado, 44
Finaliza una estructura, 143
Flujo de datos, 343
Friend, 419
Función de programación, 95
Funciones intrínsecas, 246
Function points, 106

g

Garbage In, Garbage Out, 56
Generación de datos, 11
Generación, 26
Generadores de código, 27
GIGO, 56

h

Hardware, 5
Herencia, 405, 417

i

Identidad, 407
ILINK32.EXE, 337
Imaginación, 8
Importar paquetes, 370
Incompatibilidad, 19
Información, 8
Informática, 12
Information Technology Infraestructure Library, 103
Inheritance, 405
Inherits, 418
Iniciando una estructura, 143
Input, 218
Instanciación, 403
Instrucción de asignación, 218
Instrucción de bucle, 218
Instrucción de decisión, 218
Instrucción de entrada, 218
Instrucción de llamado, 218
Instrucción de salida, 218
Interfaz, 108
Item, 150
Item, 61
ITIL, 103

j

J2SE, 368
Java 2 Plataform Standard Edition, 368
Java Development Kit, 368
Java Foundation Class, 369
Java Virtual Machine, 369
JAVA.EXE, 369
JAVAC.EXE, 369
JCreator, 369
JDK, 368
JDN, 57
JFC, 369
Julian Day Number, 57
Just in Time Compiler, 271
JVM, 369

l

Lector de datos bufferizado, 375
Lenguaje máquina, 20
Lenguajes de programación, 20
Line, 180
Línea de comandos, 216
Lógica de programación, 42

Lógica, 37
Loop limit, 180
Loop, 146
Loop, 218

m

Macroinstrucciones, 27
Macromedia DreamWeaver, 271
Main, 213
Manejador de entrada de datos, 375
Marco de trabajo, 269
Master console, 213
Members, 404
Memoria RAM, 4
Memory Address, 4
Menu bar, 213
Menu bar, 215
Métodos, 404
Microprocesador, 4
Microsoft .NET Framework Redistributable Package, 270
Microsoft .NET Framework SDK, 270
Microsoft .NET Framework, 270
Microsoft Intermediate Language, 282
Microsoft Visual C++, 337
Microsoft Visual Studio.NET, 271
Miembros, 404
Miniespecificación, 152
Modelo de Cascada, 95
Modularidad, 404
Módulo,
MSIL, 282
Muestra de datos, 142
MustInherit, 418
MustOverride, 420
MyBase, 421
MyClass, 421

n

NetBeans, 369
Nivel, 28
Non case-sensitive, 273
NotInheritable, 418
NotOverridable, 420

o

Objetivo de una interfaz, 108
Objetos, 30, 403
Opcionalidad, 58
Operación de asignación, 140
Operaciones a nivel dato, 139
Operador de asignación, 78
Operadores aritméticos, 76
Operadores comparativos, 78
Operadores lógicos, 79
Operadores, 76
Orientación, 29
Output, 218
Overloads, 421
Overridable, 420
Overrides, 421

p

Packages, 369
Página de códigos, 7
Palabras reservadas, 20
Paquetes de datos, 18
Paquetes, 369
Parallel mode, 180
Parse, 310
Parser, 25
Petición de datos, 141
Plataforma operativa, 5
Polimorfismo, 406, 422
Polimorfismo basado en herencia, 422
Portabilidad, 28
Posiciones de memoria, 4
Posiciones, 68
Precedencia explícita, 82
Precedencia implícita por categoría, 81
Precedencia implícita por operador, 81
Precedencia implícita, 81
Precedencia posicional, 82
Precedencia, 80
Predefined Process, 179
Preferencia, 80
Premisa mayor, 38
Premisa menor, 38
Premisas, 38
Private, 419

Preparation, 179
Procedimiento de evento, 29, 409
Procedimiento definido por el usuario, 410
Procedimiento principal del programa, 246
Procedimientos externos, 246
Procesamiento de datos, 11
Procesar, 37
Proceso de razonar, 37
Proceso, 42
Process, 179
Programación orientada a objetos, 403
Programa de cómputo, 43
Programa ejecutable, 24
Programas aplicativos, 5
Properties, 404
Propiedad, 404, 411
Propiedad de identidad, 408
Proposición, 37
Propósito, 28
Prototipo, 109
Pruebas de escritorio, 163
Puntos de función, 106
Puntos de interrupción, 244

r

Random Access Memory, 4
Rapid Algoritmic Prototyping Tool for Ordered Reasoning, 210
Raptor, 210
Razonar, 37
Recopilación de datos, 11
Reglas de negocios, 68, 410
Requerimientos del cliente, 103
Results, 216

s

Salida por falso, 240
Salida por verdadero, 240
Sangría, 144
Secuencias de números binarios, 18
Selection condition, 240
Selection, 218
Semántica, 21
Serie continua de valores, 66
Serie discontinua de valores, 67
Serie lógica de valores, 67
Sharp Develop, 271
Signatura, 410
Silogismo, 37
Símbolo Anotación, 180
Símbolo Conector, 180
Símbolo Datos, 179
Símbolo de muestra de datos, 143
Símbolo Decisión, 179
Símbolo Límites de bucle, 180
Símbolo Línea de flujo, 180
Símbolo Línea punteada, 180
Símbolo Modo paralelo, 180
Símbolo Preparación, 179
Símbolo Proceso Predefinido, 179
Símbolo Proceso, 179
Símbolo Terminador, 180
Símbolo, 178
Símbololo de petición, 141
Símbolos especiales, 20
Simulación, 109
Sintaxis, 21
Síntesis, 25
Sistema binario, 18
Sistema operativo, 5
Sistema, 3
Sistemas de información, 5
Software, 5
Stream, 343
Subcharts, 246
Subconjuntos, 66
Subdiagramas, 246
Subíndice, 150
Subíndice, 61
Symbols area, 214

t

Terminator, 180
Términos mnemotécnicos, 26
Tiempo de diseño, 211
Tiempo de ejecución, 211
Tipo de dato, 62
Toolbar, 213

u

UDF, 410
UNICODE, 7
Unidad aritmético lógica, 4
Unidad central de procesamiento, 4
Unidades de funcionalidad, 106, 404
Unidades de medida para los bits, 6
United States Air Force Academy, 210
USAFA, 210
User defined function, 410

v

Valor de verdad, 38
Valor, 58
Valor, 8
Valores BLOB, 58
Valores booleanos, 57
Valores cadena, 57
Valores fecha, 57
Valores lógicos, 58
Valores nulos, 58
Valores numéricos, 57
Variable, 59
vbc.exe, 271
Ventana principal, 213

w

Watch window, 214
Waterfall model, 95
Workspace, 214

Juegos, Revistas, Cursos, Software, Sistemas Operativos, Antivirus y más … Gratis para el Conocimiento...!

www.detodoprogramas.com

Visítanos y compruébalo

DETODOPROGRAMAS.COM

Material para los amantes de la Programación Java, C/C++/C#,Visual.Net, SQL, Python, Javascript, Oracle, Algoritmos, CSS, Desarrollo Web, Joomla, jquery, Ajax y Mucho Mas…

www.detodoprogramacion.com

Visitanos

Libros Universitarios, Contabilidad, Matemáticas, obras literarias, Administración, ingeniería y mas…