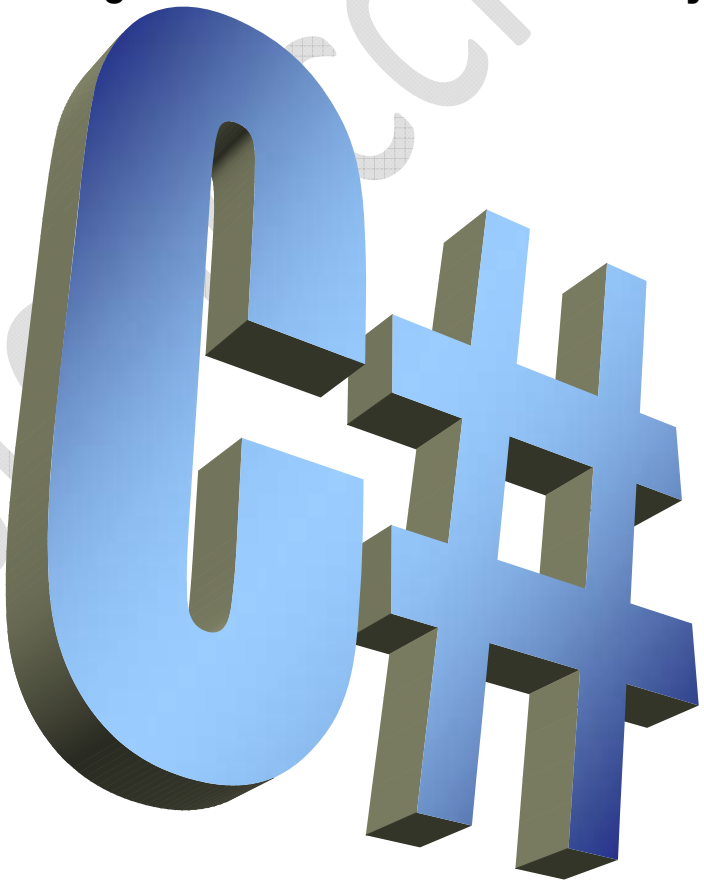
Programación Orientada a Objetos en C#:

Apruébala YA!!!

Programación Orientada a Objetos en:



Apruébala YA!!!

Be free... I'm free, I use Linux... Close your Windows, open your mind!!!

Editorial Mi-K-sita

7ª Edición (en español)

Ejemplares impresos: 1

Derechos reservados

Queda totalmente aprobada cualquier copia parcial o completa de la presente guía, aún sin el consentimiento de su autor.

© Copyleft 2008 by Fausto Iocchi

Ing. Fausto Iocchi

Introducción

La presente guía no es más que eso: **una guía** para el seguimiento de la materia *Programación Orientada a Objetos*, en donde se pretende llevarlo de la mano por un recorrido básico de la materia (recuerde además de que el profesor es un *facilitador*, mostrándole el camino que usted, como *participante*, debe seguir… lo demás lo debe hacer usted mismo), no coartando con esto que usted lea otros textos acerca del mismo… es más, se le invita a que lo haga.

Se sobreentenderá acá que ya usted cuenta con los conocimientos teóricos de la materia: conceptos tan básicos como el Juicio, el Pensamiento, el Pensar, Silogismos, etc., conceptos estos que si bien se los sabe *al caletre* vendría bien si intentase, en su tiempo libre, tratar de entenderlos y cavilar al respecto. Quizá piense que para esta materia no es indispensable estos conocimientos, pero… ¿quién sabe?

En los dos primeros capítulos de esta guía estaremos repasando los conceptos básicos de la Lógica Formal, como por ejemplo las operaciones de las que dispone la lógica proposicional y las tablas de verdad. Esto se hace ya que la lógica proposicional es la más antigua y simple de las formas de lógica. Utilizando una representación primitiva del lenguaje, permite representar y manipular aserciones sobre el mundo que nos rodea. La lógica proposicional permite *el razonamiento*, a través de un mecanismo que primero evalúa sentencias simples y luego sentencias complejas, formadas mediante el uso de conectivos proposicionales, como por ejemplo "Y" y "O". Este mecanismo determina la veracidad de una sentencia compleja, analizando los valores de veracidad asignados a las sentencias simples que la conforman.

El tema de los algoritmos (crucial para esta materia), aunque no se tocará en esta guía, es importante que se domine, ya que esta es una de las bases fundamentales para la elaboración de programas de computadoras… se sobreentenderá que ya usted posee cierto nivel de pericia en cuanto a este tema, por lo que el profesor podrá, en algún momento, evaluárselo.

Los siguientes capítulos contienen información primero, teórica, tocando toda la parte de la historia de la computación, conceptos básicos de los sistemas operativos, etc. Posteriormente, se irá introduciendo al mundo de la programación orientada a objetos (POO).

En la presente guía se tomará el lenguaje C# para ejemplificar y explicar los conceptos de la POO, sin decir esto que dichos conceptos no puedan ser aplicados en otros lenguajes que cumplan con los requerimientos de la POO… recuerde que este es una asignatura en la cual usted aprenderá los conceptos de la POO y su aplicación, más no es un curso de programación.

El último capítulo tiene una serie de ejercicios propuestos. Si usted dispone del tiempo necesario, reúnase en grupo, a ser posible (o incluso en la soledad de su habitación) y resuélvalos, para que de esta manera ir obteniendo la pericia necesaria para poder aprobar esta materia.

Acá se sobreentenderá que ya usted cuenta con un lenguaje de programación denominado **Microsoft Visual Studio .NET 2003** (como mínimo), instalado en el computador en el cual realizará los ejemplos y prácticas recomendadas en esta guía. En caso de no contar con esto, se le recomienda su adquisición e instalación a la brevedad posible, ya que sin él no podrá hacerle seguimiento a los diversos ejercicios acá explicados.

Espero que la presente le sirva de ayuda… Éxitos.

Ing. Fausto Lochi

Unidad I

Objetivo:

Previo repaso y lectura acerca de la Lógica Proposicional, el participante conformará una estructura teórica y práctica que le permita la comprensión de esta misma, así como del razonamiento lógico.

Contenido:

1. Lógica Proposicional
   1.1. Simbolización de proposiciones
   1.2. El Cálculo Proposicional como un Sistema Axiomático
      1.2.1. Signos primitivos
      1.2.2. Reglas Formativas
      1.2.3. Signos Definidos
      1.2.4. Razonamiento Lógico

Actividades del participante:

- Realizar lectura previa sobre el tema, utilizando la presente guía y otros medios bibliográficos.
- Interactuar en clase.
- Discutir en grupo.
- Realizar las actividades propuestas en clase.

Evaluación multidireccional:

- Evaluación formativa: a través de las discusiones de los conocimientos e intervenciones en clase.
- Evaluación sumativa: a través de un examen práctico y/o parcial.

Lógica Proposicional

1. Lógica Proposicional

Como ya se debe saber, la *Lógica Proposicional* es la más antigua y simple de las formas de lógica. Utilizando una representación primitiva del lenguaje, permite representar y manipular aserciones sobre el mundo que nos rodea. La lógica proposicional permite el razonamiento, a través de un mecanismo que primero evalúa sentencias simples y luego sentencias complejas, formadas mediante el uso de conectivos proposicionales, por ejemplo "Y" y "O". Este mecanismo determina la veracidad de una sentencia compleja, analizando los valores de veracidad asignados a las sentencias simples que la conforman.

La lógica proposicional pretende estudiar las frases declarativas simples (enunciados o proposiciones) que son los elementos básicos de transmisión de conocimiento humano.

1.1. Simbolización de proposiciones

Cada proposición tiene una forma lógica a la cual se le da un nombre. Se distinguen dos tipos de proposiciones: simples y compuestas. Una proposición se denomina simple cuando en ella no interviene ninguna conectiva lógica o término de enlace (y, o, no, si...entonces..., si y sólo si). Si se juntan una o varias proposiciones simples con un término de enlace, se forma una proposición compuesta.

Los términos de enlace, "y", "o", "si... entonces...", "si y sólo si" se usan para unir dos proposiciones. En cambio el término de enlace "no" se agrega a una sola proposición. Por ejemplo:

Hoy es jueves Hay clases de matemáticas

Ambas proposiciones son simples. Con estas proposiciones se pueden construir proposiciones compuestas tales como:

Ing. Fausto Toschi

Hoy es jueves y hay clases de matemáticas
Hoy es jueves o hay clases de matemáticas
Si hoy es jueves entonces hay clases de matemáticas
Hoy no es jueves

Para representar las proposiciones se utilizan letra latinas mayúsculas tales como P, Q, R, etc. Por ejemplo, sea:

P: Hoy es jueves
Q: Hay clase de matemáticas

Luego la proposición:

Hoy es jueves y hay clase de matemáticas

se podría simbolizar así:

P y Q

En el lenguaje corriente se utiliza también la palabra "pero" o una "," en vez del término de enlace "y". Ejemplo:

Fui a la feria, pero no hice compra alguna
Inés está enferma, el martes iré a visitarla

En el siguiente ejemplo se usa el término de enlace "o":

Es tarde o está muy oscuro

Otro giro de "o" es:

O es tarde o está muy oscuro

En este último caso las dos "o" son parte del mismo término de enlace.

Cuando se usa el término de enlace: si,...entonces.... se obtiene la siguiente forma:

Si R entonces S

Ejemplo:

Si madrugo entonces llego temprano

En este ejemplo puede suprimirse la palabra "entonces" y reemplazarse por una "," así:

Si madrugo, llego temprano

Cuando la palabra "no" se encuentra en el interior de una proposición simple, puede pasar inadvertida, pero se trata de una proposición compuesta. Ejemplo:

El día no está caluroso

Puede presentarse como:

No ocurre que el día esté caluroso

También se usan símbolos para representar los términos de enlace, así:

Para la "y" se utiliza el símbolo (^).
Para la "o" se utiliza el símbolo (v).
Para el "no" se utiliza el símbolo (¬).
Para el "si,...entonces..." se utiliza el símbolo (→).
Para el "si y sólo si" se utiliza el símbolo (←→).

Cuando una proposición compuesta utiliza el término de enlace "y" es una conjunción. Si el enlace se hace mediante la conectiva "o" es una disyunción. Si se usa el término "no" es una negación. Cuando la conectiva es "si...entonces..." es una proposición condicional, y si utiliza "si y sólo si" se tiene un bicondicional.

Ing. Fausto Pocchi

En proposiciones que tienen más de un término de enlace es preciso indicar la manera de agruparse, pues distintas agrupaciones pueden tener distintos significados. En el lenguaje corriente, las agrupaciones se presentan de acuerdo a la colocación de ciertas palabras o mediante la puntuación. En lógica, la agrupación se indica por medio de paréntesis. Ejemplo de esto es:

O los soldados encontraron cerrado el paso, o si temieron un ataque enemigo, se refugiaron en las montañas. Este texto se simboliza de la siguiente forma:

P: Los soldados encontraron cerrado el paso
Q: Los soldados temieron un ataque enemigo
R: Los soldados se refugiaron en las montañas

La proposición compuesta es:

$P \vee (Q \rightarrow R)$

La cual tiene un sentido distinto de la proposición:

$(P \vee Q) \rightarrow R$

Cuando no hay lugar a ambigüedades, pueden omitirse los paréntesis y se adopta una convención con respecto a la dominancia relativa de los diversos conectivos. La convención es:

"$\leftrightarrow$" y "$\rightarrow$" dominan a "$\wedge$" y "$\vee$"

así:

$S \leftrightarrow P \vee R$ significa $S \leftrightarrow (P \vee R)$

$P \rightarrow Q \wedge R$ significa $P \rightarrow (Q \wedge R)$

Con esta convención, no está claro lo que significa, por ejemplo:

$P \wedge Q \vee R$ ó $P \leftrightarrow Q \rightarrow R$

Aquí, es necesario usar paréntesis para aclarar, en el primer caso, si se trata de:

$(P \wedge Q) \vee R$ ó $P \wedge (Q \vee R)$

y en el segundo caso, diferenciar entre:

$(P \leftrightarrow Q) \rightarrow R$ y $P \leftrightarrow (Q \rightarrow R)$

1.2. El Cálculo Proposicional como un Sistema Axiomático

1.2.1. Signos primitivos

Son signos primitivos aquellos que utilizamos para conformar una proposición. Por ejemplo:

- *Letras latinas, tanto en minúsculas como mayúsculas*.
- *Signos lógicos*: "¬", "v", "^".
- *Signos de puntuación*: "(" y ")".

De las sucesiones de signos que es posible construir, hay específicamente unas que tienen sentido dentro de la teoría. Tales sucesiones se denominan, términos y fórmulas. Los términos se identifican con los objetos de la teoría y las fórmulas expresan relaciones entre los objetos. La especificación de los términos y las fórmulas, se hace a través de las siguientes reglas:

1.2.2. Reglas Formativas

RF1: Cualquier letra es un término.
RF2: Si R es una fórmula, entonces ¬R es una fórmula, la cual se denomina negación de R.
RF3: Si P y Q son fórmulas, entonces P v Q es una fórmula la cual se denomina disyunción lógica de P y Q.

Ing. Fausto Iocchi

Nota: En las RF2 y RF3, las letras mayúsculas se usan para designar fórmulas, no corresponden a signos del lenguaje.

1.2.3. Signos Definidos

Una vez establecidas las reglas de formación de fórmulas se pueden introducir abreviaciones con el fin de simplificar la escritura. Estas abreviaciones son el objeto de las definiciones matemáticas.

Definición: Sean R y S fórmulas, entonces:

- La fórmula ¬(¬R v ¬S) se denota abreviadamente como R ^ S y se llama *conjunción lógica* de R y S, la cual se lee "R y S.
- La fórmula ¬ R v S se denota como R → S y se llama *condicional* de R y S. La figura lógica del condicional, responde a conectar dos proposiciones mediante el esquema "si..., entonces...". Para leer una proposición de la forma R → S, se puede usar algunas de las siguientes expresiones:

Si R entonces S R es suficiente para S S es necesario para R S siempre que R R sólo si S

A la fórmula R se le llama *antecedente*, y a la fórmula S *consecuente*. Cuando el condicional es verdadero se dice que existe implicación y en este caso se lee la expresión como:

R implica S

la cual se denota:

R → S

- La fórmula (R → S) ^ (S → R) se denota por R ←→ S y se llama bicondicional de R y S. Esta expresión se puede leer (o entender) de varias maneras, entre ellas:

R si y sólo sí S R es suficiente y necesario para S

Cuando el bicondicional es verdadero, se dice que hay *equivalencia.* En este caso se lee:

R equivale a S

y se denota:

R ⇔ S

Nota: Los criterios para decidir sobre la verdad del condicional y el bicondicional se verán más adelante.

1.2.4. Razonamiento Lógico

Deducción. La deducción lógico matemática consiste en lo siguiente: A partir de una serie de fórmulas admitidas como ciertas, y denominadas *axiomas, hipótesis o premisas*, se obtiene otra fórmula llamada *conclusión* o *tesis*, mediante la aplicación de reglas lógicas precisas. El proceso mediante el cual se pasa de las hipótesis (premisas) a la tesis, recibe el nombre de *demostración*.

Un *teorema* es una fórmula que figura dentro de una demostración. Es decir, un teorema es una fórmula que es o bien un axioma, o bien, una consecuencia de éste.

Una fórmula se dice que es *falsa* si su negación es un teorema. Una teoría es *contradictoria* cuando se tiene una fórmula R que es verdadera y falsa a la vez. Esto es: R y ¬R son teoremas de la teoría.

Para demostrar que una fórmula C es un teorema se desarrolla el siguiente

Ing. Fausto Iocchi

proceso:

- Se enuncian los axiomas de la teoría. Para la lógica proposicional se establecen cuatro axiomas, que son:

A1. *Axioma de idempotencia***.** Sea P una fórmula, entonces, la fórmula:

P v P ➔ P es un axioma

A2. *Axioma de adjunción***.** Sean P Y Q fórmulas, entonces, la fórmula:

P ➔ P v Q es un axioma

A3. *Axioma de conmutatividad***.** Sean P, Q y R fórmulas, entonces, la fórmula:

P v Q ➔ Q v P es un axioma

A4*. Axioma de adición***.** Sean P, Q y R fórmulas, entonces, la fórmula:

(P ➔ Q) ➔ (R v P ➔ R v Q) es un axioma

- Se fijan las "reglas lógicas" que permiten deducir dicha fórmula a partir de los axiomas. Estas reglas son llamadas reglas de validez (RV) y son las siguientes:

RV1: Dadas las fórmulas R y S; si R ➔ S y R son verdaderas, entonces S es verdadera.
RV2: Si R y S son fórmulas equivalentes, se puede sustituirla una por la otra en cualquier parte del proceso demostrativo.

- Se hace una demostración de la fórmula C, que consiste en obtener a C como última fórmula de la lista, por aplicación reiterada de RV1 y RV2.

Ing. Fausto Iocchi

Unidad II

Objetivo:

Previo repaso y lectura acerca de la Tabla de Verdad y la unidad anterior, el participante formalizará expresiones y razonamientos lógicos, los evaluará por completo e inferirá sobre sus resultados.

Contenido:

2. Tablas de Verdad.
   2.1. Operadores Lógicos.
      2.1.1. Negación.
      2.1.2. Disyunción.
      2.1.3. Conjunción.
      2.1.4. Condicional.
      2.1.5. Bicondicional.
   2.2. Propiedades y Postulados.
      2.2.1. Conmutativa.
      2.2.2. Distributiva.
      2.2.3. Elemento Neutro.
      2.2.4. Elemento Inverso.
   2.3. Problemas resueltos.

Actividades del participante:

- Realizar lectura previa sobre el tema, utilizando la presente guía y otros medios bibliográficos.
- Interactuar en clase.
- Discutir en grupo.
- Realizar las actividades propuestas en clase y en esta unidad.

Evaluación multidireccional:

- Evaluación formativa: a través de las discusiones de los conocimientos e intervenciones en clase.
- Evaluación sumativa: a través de un examen práctico y/o parcial.

Tablas de Verdad

2. Tablas de Verdad

Estas tablas pueden construirse haciendo una interpretación de los signos lógicos ¬, ^, v, →, ←→, así como: no, y, o, si…entonces, sí y sólo si, respectivamente. La interpretación corresponde al sentido que estas operaciones tienen dentro del razonamiento.

Puede establecerse una correspondencia entre los resultados de estas tablas y la deducción lógico matemática. En consecuencia, las tablas de verdad constituyen un método de decisión para chequear si una proposición es o no un teorema.

Para la construcción de la tabla se asignará el valor 1 (uno) a una proposición cierta y 0 (cero) a una proposición falsa. Esta ha sido la manera universal en que se ha expresado, y todo esto gracias a George Boole (1815-1864), quién fue la persona que ideó el álgebra que lleva su nombre. El presentó el primer tratamiento sistemático de la lógica, y para ello desarrolló un sistema algebraico, conocido ahora como Álgebra de Boole. Además de sus aplicaciones al campo de la lógica, el álgebra de Boole ha tenido dos aplicaciones importantes: el tratamiento de conjuntos mediante las operaciones de unión e intersección que ha servido de base a la teoría de la probabilidad y el diseño de circuitos digitales combinacionales.

Aunque en el álgebra de Boole en realidad se utilizan otros símbolos para los operadores, vamos a mantener acá los símbolos de las proposiciones lógicas. Lo que sí se tomó acá de dicha álgebra son los valores 0 y 1 para la notación de algo Falso o Cierto, respectivamente. Esto se hace para evitar confusiones más adelante.

2.1. Operadores Lógicos

Existen unas cinco operaciones básicas que operan sobre las proposiciones lógicas, como ya lo hemos visto anteriormente. Estudiemos ahora la Tabla

Ing. Fausto Inochi

de Verdad para cada uno de ellos.

2.1.1. **Negación:** El valor de verdad de la negación es el contrario de la proposición negada.

P	¬P
1	0
0	1

2.1.2. **Disyunción:** La disyunción solamente es falsa si lo son sus dos componentes. Esta operación es ampliamente conocida con el nombre del operador "O".

P	Q	P v Q
1	1	1
1	0	1
0	1	1
0	0	0

2.1.3. **Conjunción:** Solamente si las componentes de la conjunción son ciertas, la conjunción es cierta. Esta operación es conocida también bajo el nombre de "Y".

P	Q	P ^ Q
1	1	1
1	0	0
0	1	0
0	0	0

2.1.4. **Condicional:** El condicional solamente es falso cuando el antecedente es verdadero y el consecuente es falso. De la verdad no se puede seguir la falsedad.

P	Q	P → Q
1	1	1
1	0	0
0	1	1
0	0	1

2.1.5. **Bicondicional:** El bicondicional solamente es cierto si sus componentes tienen el mismo valor de verdad.

P	Q	P ←→ Q
1	1	1
1	0	0
0	1	0
0	0	1

Se denomina *tautología* una proposición que es cierta para cualquier valor de verdad de sus componentes. Por tanto, la *última columna* de su tabla de verdad estará formada únicamente por unos (o sea, en otras palabras, la última columna deberá contener, en todas sus casillas, el valor de cierto).

Contradicción es la negación de una tautología. Por tanto es una proposición falsa cualesquiera sea el valor de verdad de sus componentes. La *última columna* de la tabla de verdad de una contradicción estará formada únicamente por ceros (o sea, absolutamente todos tendrán el valor de falso).

Ing. Fausto Locchi

2.2. Propiedades y Postulados

Existen ciertas propiedades (convertidas en postulados), que se cumplen dentro de las operaciones que se pueden realizar sobre las proposiciones lógicas (así como también en el álgebra de Boole). Estas propiedades se pueden resumir en lo siguiente:

2.2.1. **Conmutativa:**
P ^ Q = Q ^ P
P v Q = Q v P

2.2.2. **Distributiva:**
P ^ (Q v R) = (P ^ Q) v (P ^ R)
P v (Q ^ R) = (P v Q) ^ (P v R)

2.2.3. **Elemento Neutro:**
P ^ 1 = P
P v 0 = P

2.2.4. **Elemento Inverso:**
P ^ ¬P = 0
P v ¬P = 1

2.3. Problemas resueltos

Trate de realizar los siguientes ejercicios a manera de práctica. Los ejercicios a evaluar en los exámenes serán de una tónica muy parecida.

2.3.1. Formalizar las siguientes expresiones:

a) q si p
b) p pero q
c) como mínimo p
d) p no obstante q
e) q necesario para p
f) q suficiente para p
g) p a pesar de q
h) no p a menos que q
i) p sólo si q
j) p sin embargo q
k) p suficiente para q
l) p siempre que q
m) a veces p, siempre q
n) p a no ser que q

Solución:

a) $p \rightarrow q$
b) $p \wedge q$
c) p
d) $p \wedge q$
e) $p \rightarrow q$
f) $q \rightarrow p$
g) $p \wedge q$
h) $p \rightarrow q$
i) $p \rightarrow q$
j) $p \wedge q$
k) $p \rightarrow q$
l) $q \rightarrow p$
m) $(p \vee \neg p) \wedge q \equiv q$
n) $\neg q \rightarrow p$

2.3.2. Formalizar los siguientes razonamientos:

"Si el resultado obtenido es superior al previsto en 5 unidades, será debido a no haber realizado el proceso a la temperatura adecuada o a la existencia de errores en los cálculos finales."

"El análisis realizado, innecesario si nos dejamos llevar por la precipitación, se torna necesario si nos paramos a reflexionar sobre el mensaje que se pretende transmitir."

"El cáncer no logrará curarse a no ser que se logre determinar su

Ing. Fausto Iocchi

causa y se consiga encontrar fármacos adecuados o bien para prevenirlo o para curarlo."

Solución:

p = Resultado obtenido menor al previsto en 5 unidades.
q = Haber realizado el proceso a la temperatura adecuada.
r = Existencia de errores en los cálculos finales.

$(\neg q \vee r) \rightarrow p$

p = Análisis realizado es necesario.
q = Nos dejamos llevar por la precipitación.
r = Nos paramos a reflexionar sobre el mensaje que se pretende transmitir.

$(q \rightarrow \neg p) \wedge (r \rightarrow p)$

p = El cáncer logrará curarse.
q = Se logra determinar su causa.
r = Se consigue encontrar fármacos adecuados para prevenirlo.
s = Se consigue encontrar fármacos adecuados para curarlo.

$\neg\left(q \wedge (r \vee s)\right) \rightarrow \neg p \equiv p \rightarrow \left(q \wedge (r \vee s)\right)$

2.3.3. Formalizar utilizando las conectivas {¬, ∨, ∧} el siguiente enunciado: "Si p entonces q y, en caso contrario, si p1 entonces q1 y, en caso contrario r1."

Solución:

$(p \rightarrow q) \wedge \left(\neg p \rightarrow \left((p1 \rightarrow q1) \wedge (\neg p1 \rightarrow r1)\right)\right)$

2.3.4. Formalizar el siguiente enunciado donde hemos notado "si p entonces q y en caso contrario r" de la siguiente forma:

$p \rightarrow q;r$

Normalizar dicho enunciado.

Solución:

$(p \rightarrow q) \wedge (\neg p \rightarrow r)$

Forma normal conjuntiva: $(\neg p \vee q) \wedge (p \vee r)$

2.3.5. Compruébese si los siguientes razonamientos son correctos o no:

a. "Si Antonio ganó la carrera, entonces Baltasar o Carlos fueron los segundos. Si Baltasar fue segundo, entonces no ganó Antonio. Si Demetrio fue segundo, no lo fue Carlos. Antonio ganó la carrera. Por tanto, Demetrio no fue segundo".
b. "No llora, ríe. Si no llora, ríe sólo si tiene un juguete. Nunca tiene un juguete cuando se está riendo si no come un caramelo. Luego come un caramelo."
c. "Juan quiere a María si y sólo si María quiere a Juan y promete casarse con él. María no quiere a Juan si Juan no quiere a María. María promete casarse con Juan si y sólo si Juan promete casarse con María. Por tanto, Juan quiere a María y María no quiere a Juan".
d. "Si ha nevado será difícil conducir. Si no es fácil conducir llegaré tarde si no salgo temprano. Ha nevado. Luego saldré temprano."
e. "Si no llueve salgo al campo. Si salgo al campo respiro. Por tanto, respiro si y sólo si no llueve."
f. "Si un monte se quema algo tuyo se quema. Algo tuyo se quema si y sólo si eres descuidado. Si eres descuidado no mereces que te feliciten. Por tanto si no mereces que te feliciten entonces es que un monte se quema."
g. "El Ministro Popular de Economía y Hacienda ha hecho las siguientes declaraciones:
A la prensa: "Si los impuestos suben, la inflación bajará si y sólo si no se devalúa el Bolívar."
A la radio: "Si la inflación baja o si el Bolívar no se devalúa, los

Ing. Fausto Iocchi

impuestos no subirán."
A la tele: "O bien baja la inflación y se devalúa el Bolívar, o bien los impuestos deben subir."
Como consecuencia, publica un informe en el que asegura: "Los impuestos deben subir, pero la inflación bajará y el Bolívar no se devaluará."
¿Fue consecuente con sus declaraciones a los medios de comunicación?".

h. "Es suficiente whisky para que chocolate. Chocolate si y solo si jamón. No ginebra a menos que chocolate. Whisky. ¿Es posible afirmar: (1) ¿que bebió ginebra? (2) ¿que no tomó chocolate?"
i. "Si no especifico las condiciones iniciales mi programa no comenzará. Habré programado un ciclo infinito solo si mi programa no termina. Basta que el programa no comience o no finalice para que falle. De ahí que sea necesario no solamente especificar las condiciones iniciales sino también no programar un ciclo infinito para que el programa no falle."
j. "Si 25 divisiones son suficientes, el general ganará la batalla; por otra parte, o se suministran 3 alas de apoyo aéreo táctico, o el general no ganará la batalla. Además, no es cierto que sean suficientes 25 divisiones y que se vayan a suministrar 3 alas de apoyo aéreo táctico. Conclusión: no son suficientes 25 divisiones."

Solución:

a. SI es válido, Antonio ganó la carrera.
b. ES válido, come un caramelo.
c. El razonamiento NO es válido, ya que puede darse el caso de que ninguno de los dos quiera al otro, y las premisas serían ciertas, pero la conclusión Falsa.
d. El razonamiento NO es válido porque puede darse el caso de NO salir temprano y llegar tarde habiendo nevado y siendo difícil conducir. Cumpliéndose todas las premisas.
e. NO es válido, puedo salir al campo, lloviendo y respirar. Luego no se deduce que respire si y sólo si no llueve.
f. NO es válido.
g. NO es correcto.
h. NO se puede deducir ninguno de los dos, ni que bebiese Ginebra ni que no tomase chocolate.
i. El razonamiento ES correcto.
j. El razonamiento ES correcto.

2.3.6. Le digo a un amigo: "Cuando salgo sin paraguas, llueve. Cuando está despejado, no llueve. Según el hombre del tiempo, mañana estará despejado o hará niebla. De todos modos saldré sin paraguas". Entonces mi amigo responde: "Entonces mañana, además de llover, habrá niebla". ¿Cómo lo supo?

Solución:

Ambas expresiones se deducen lógicamente suponiendo que todas las frases sean Verdaderas, se puede comprobar que sólo existe esa posibilidad.

2.3.7. Don Juan Tenorio, hizo las siguientes declaraciones, con respecto a las doncellas Inés, Juana y María, que le costaron la vida. ¿Quién o quiénes son las asesinas?: "Amo a la última de las tres. Si amo a Inés pero no a María, entonces también amo a Juana. O amo a María y a Juana o no amo a ninguna. Si amo a María, entonces amo a Inés". (se supone que la asesina era aquélla a la que Don Juan no amaba).

Solución:

Ninguna de ellas era la asesina, pues las amaba a las tres.

2.3.8. En un juicio el fiscal argumenta: "Si el acusado es culpable, entonces tenía un testigo". A ello, el abogado defensor respondió inmediatamente: "Eso es falso". El acusado decidió cambiar de abogado defensor. ¿Es lógica la decisión tomada por el acusado? ¿Por qué?

Ing. Fausto Roschi

Solución:

Si es Falsa la sentencia "Si el acusado es culpable entonces tenía un testigo", por la tabla de verdad de la implicación, el antecedente es Verdadero y el consecuente Falso, luego el antecedente es Verdadero, es decir, el acusado sería culpable.

2.3.9. Analizar la coherencia lógica (no teológica) del siguiente razonamiento: "Si Dios existe es todo amor y omnipotencia. Si Dios es incapaz de erradicar el sufrimiento del mundo entonces no es omnipotente. Dios no es amor o está dispuesto a erradicar el sufrimiento del mundo. Dios es capaz de erradicar el sufrimiento del mundo y está dispuesto a ello solo si no existe sufrimiento en el mundo. Existe sufrimiento en el mundo. Por tanto: Dios no existe".

Solución:

El razonamiento es correcto en términos lógicos.

2.3.10. "Si la Bella Durmiente despierta, los habitantes del castillo también lo harán. Si el príncipe la besa, despertará. El príncipe la besará si está de buen ver. O la besa o no se despierta nadie. Como esto es un cuento, la princesa, a pesar de llevar 100 años dormida sigue de muy buen ver. Si la princesa se despierta se casarán, vivirán felices y comerán perdices si no estamos en veda. Estamos en veda". ¿Se casan? ¿Son felices? ¿Comen perdices?

Solución:

Se casan, viven felices, pero no comen perdices porque estamos en veda.

2.3.11. Los dos carteles siguientes están colgados respectivamente a la puerta de las habitaciones 1 y 2. Uno de los carteles dice la verdad y el otro cartel miente. Sabiendo que en la misma habitación no puede haber una dama y un tigre, y que puede haber dos damas y dos tigres, se pide decidir lógicamente qué puerta se debe abrir para liberar a la dama (si es que existe). Ambas habitaciones están ocupadas.

CARTEL 1: En esta habitación hay una dama y en la otra un tigre.
CARTEL 2: En una de estas habitaciones hay una dama y en una de estas habitaciones hay un tigre.

Solución:

La única forma de que un cartel diga la verdad y otro mienta es que el Cartel 1 mienta y el Cartel 2 diga la verdad. Con lo cual habría un tigre en la habitación 1 y una dama en la habitación 2.

2.3.12. Igual al problema anterior, pero ahora suponiendo que los dos carteles siguientes dicen ambos la verdad o mienten ambos. Deducir en qué habitación hay una dama, sabiendo, como antes, que puede no haberla.

CARTEL 1: Al menos en una de estas habitaciones hay una dama.
CARTEL 2: Hay un tigre en la otra habitación.

Solución:

La única posibilidad es que los dos carteles digan la verdad y habría una dama en la habitación 2 y un tigre en la habitación 1.

2.3.13. Discurso sobre los estudios de Sistemas en clase de Lógica: "Señoras, señores, buenas tardes: Es hora de que recapacitemos sobre los estudios de sistemas en vísperas de la graduación en nuestra Universidad. Se sabe que si las computadoras hablasen los sistémicos (Ing. En sistemas) no existirían. Por otra parte, en la última reunión del Consejo de Universidades, éste afirmó que: "...la Universidad graduará sistémicos mientras las computadoras no hablen..."; afirmación que nos parece muy correcta, si bien lo

Ing. Fausto Jocchi

cierto es que las computadoras no hablan pero los sistémicos existen. A la vista de todo ello nos preguntamos: ¿Es, por tanto, coherente que la Universidad expida títulos de Ingenieros de Sistemas en la actualidad?

Solución:

Sí, se sigue que la universidad expenda títulos de Sistemas a partir de las premisas. El razonamiento es correcto.

Ing. Fausto Iocchi

Unidad III

Objetivo:

Previo repaso, lectura e investigación, el participante recordará la historia de la computación, así como enumerará las diferentes generaciones del computador y los sistemas operativos. También reconocerá los diversos lenguajes de programación y relacionará sus paradigmas.

Contenido:

3. Historia de la Computación.
   3.1. Procesamiento de datos.
   3.2. Las 5 generaciones del computador.
   3.3. Computador.
   3.4. Sistemas operativos.
   3.5. Lenguajes de Programación.
   3.6. Intérpretes y compiladores.
   3.7. Paradigmas de la Programación.
   3.8. Criterios del lenguaje.

Actividades del participante:

- Realizar lectura previa sobre el tema, utilizando la presente guía y otros medios bibliográficos.
- Interactuar en clase.
- Discutir en grupo.
- Realizar las actividades propuestas en clase.

Evaluación multidireccional:

- Evaluación formativa: a través de las discusiones de los conocimientos e intervenciones en clase.
- Evaluación sumativa: a través de un examen práctico y/o parcial.

Historia de la Computación

3. Historia de la Computación

3.1. Procesamiento de datos

El origen de la computación surge de la necesidad consecuente del hombre por realizar cálculos numéricos y procesar información, que le permitiera facilitar sus tareas cotidianas. Prueba de ello son los numerosos y novedosos inventos creados que han conllevado al desarrollo de la tecnología actual. En la historia surgieron muchos inventos, nuevos y aplicables, pero sólo algunos de ellos marcaron una brecha en el desarrollo de la computación, los cuales son:

3.1.1. *Bastón con muescas*: es un instrumento de madera con forma de bastón y con hendiduras adyacentes, cuya funcionalidad se basaba en llevar el conteo de algún hecho específico, sin sistemas decimales, como lo conocemos hoy en día, sólo daba un indicio de las veces que sucedía algún evento.

3.1.2. *Ábaco Chino*: es un instrumento que permitió crear la estructura del primer sistema numérico (decimal), y con ello, poder manipular los números con mayor propiedad y precisión. Constaba con un mecanismo que permitía asignar las unidades, decenas y centenas del número que se quería representar.

3.1.3. *Blas Pascal (1.600)*: Fue un científico que logró crear una máquina sumadora a través de un sistema de engranajes mecánicos de distintos tamaños, en donde cada uno de ellos representaba una unidad específica.

3.1.4. *Gotfred Von Leibnitz (1.610)*: Tomó la idea de Pascal y logró crear una máquina, que además de sumar y restar, multiplicaba y dividía.

3.1.5. *Herman Hollerit (1.880)*: Construyó una máquina, cuya función era la de interpretar una serie de tarjetas perforadas, en donde el

Ing. Fausto Rocchi

punto de perforación significaba un estado verdadero *(true ó 1),* y el punto de no perforación falso *(false ó 0)*. Esta serie de 1's y 0's era procesada y nos daba una información específica. Estas tarjetas fueron usadas en el censo realizado en EEUU para ese año.

3.1.6. *ENIAC (1.946)*: *Electronical Numerical Integrator and Calculador,* fue el primer computador creado por el hombre en Pensilvania, compuesto por 18.000 tubos al vacío, pesaba tres toneladas y ocupaba un espacio de aproximadamente 150 mts^2. Este computador podía ejecutar 1.000 operaciones por segundo, toda una maravilla para ese entonces. Los programadores de estas macrocomputadoras debían conocer a fondo el hardware de la misma para poder manipularlas con precisión.

3.2. Las 5 generaciones del computador

A partir de la creación de ENIAC, comenzó una carrera tecnológica por mejorar los procesos de cómputo, aumentando su velocidad y compatibilidad, y por supuesto, reduciendo su tamaño y complejidad. Para ello se crearon nuevas tendencias y tecnologías que dejaron huellas en el desarrollo de la computación.

3.2.1. *Primera Generación (1.946-1.955)*: esta generación la conformaron las macrocomputadoras como ENIAC, que trabajaban en lenguaje binario y ocupaban grandes espacios físicos.

3.2.2. *Segunda Generación (1.955-1.963)*: este fue el primer salto histórico en el desarrollo de la computación. Se creó el transistor, que logró simular el comportamiento de centenares de tubos al vacío en un solo componente electrónico. Se redujo notablemente el tamaño de los computadores, el lenguaje usado era el ensamblador (nemónicos) y los dispositivos de almacenamiento estaban hechos de material ferromagnético.

3.2.3. *Tercera Generación (1.963-1.970)*: esta generación se basó en la reducción del tamaño del computador más que en el aumento de la velocidad, miniaturizando aun más los componentes actualmente utilizados.

3.2.4. *Cuarta Generación (1.970-1.985)*: esta generación representó el segundo gran salto histórico en el desarrollo de la computación. Se logró crear chips de silicio que lograban integrar a millones de transistores que reducían notablemente el tamaño del computador y aumentaba indiscutiblemente su velocidad de procesamiento. Para esta época eran accesibles a las personas y eran medianamente compatibles.

3.2.5. *Quinta Generación (Años 90)*: a partir de los 90, el desarrollo de la computación se ha enfocado mucho más en la creación de software que permitan procesar el conocimiento eficientemente, se crea la inteligencia artificial y los sistemas expertos (reconocedor de huellas, imágenes, voces, etc.). La compatibilidad entre software y hardware es total en esta generación.

3.3. Computador

Es un sistema electrónico rápido y exacto que manipula datos de entrada, los procesa y genera datos de salida según las ordenes de un programa de instrucciones almacenadas en su memoria. Está compuesto por dos bloques esenciales: *Hardware (HW),* se refiere a la parte física o arquitectura del computador, integrada por componentes electrónicos y *Software (SW)*, es la parte lógica del sistema conformada por los programas que dirigen las funciones del sistema computacional o un hardware.

3.3.1. *Arquitectura del computador*: el HW del computador está formado esencialmente por los periféricos y la CPU (unidad central de procesamiento), la cual se encarga de controlar el flujo de información. La CPU, a su vez, está dividida en tres bloques: *Unidad de memoria principal (UMP), Unidad aritmética lógica (ALU) y Unidad de Control (UC)*. Cada bloque cumple una función específica dentro de la CPU.

La unidad de control tiene como función llevar el orden de los procesos internos ejecutados en la CPU. La UC trabaja en dos

Ing. Fausto Cecchi

ciclos: ciclo de identificación y localización, y ciclo de ejecución.

La unidad aritmética lógica, se encarga de realizar todas las operaciones aritméticas (+ - * /) y comparar valores para obtener resultados de verdadero o falso.

La unidad de memoria principal es utilizada por la CPU para grabar y extraer información, ya sea en forma temporal o permanente. Existen dos tipos de memoria básicas: RAM (*Random Access Memory)*, es la llamada memoria del usuario, debido a que el programador puede leer y escribir datos en la misma. Esta memoria es temporal, ya que mantiene sus datos siempre y cuando esté conectada a un suministro de energía, su información es volátil. ROM (*Read Only Memory)*, es la memoria del fabricante, sus datos son permanentes y no pueden ser modificados mediante programación.

3.3.2. *Software del computador*: Existen tres tipos de software para el computador: de desarrollo, de aplicación y los sistemas operativos.

Sistema operativo: es el software que controla la ejecución de todas las aplicaciones y de los programas de software de sistema.

Programas de ampliación: o también llamado software de aplicación, es el software diseñado y escrito para realizar una tarea específica, ya sea personal, o de procesamiento. Aquí se incluyen las bases de datos, procesamiento de textos, hojas de cálculo electrónicas, gráficas, comunicaciones, etc., etc.

Lenguajes de programación: son las herramientas empleadas por el usuario para desarrollar programas, que luego van a ser ejecutados por la PC.

3.4. Sistemas operativos

3.4.1. *Por número de usuario*: estos se dividen en

Monousuarios: Los sistemas operativos monousuarios son aquéllos que soportan a un usuario a la vez, sin importar el número de procesadores que tenga la computadora o el número de procesos o tareas que el usuario pueda ejecutar en un mismo instante de tiempo. Las computadoras personales típicamente se han clasificado en este renglón.

Multiusuarios: Los sistemas operativos multiusuarios son capaces de dar servicio a más de un usuario a la vez, ya sea por medio de varias terminales conectadas a la computadora o por medio de sesiones remotas en una red de comunicaciones. No importa el número de procesadores en la máquina ni el número de procesos que cada usuario puede ejecutar simultáneamente.

3.4.2. *Por número de tareas*: estos se dividen en

Monotareas: Los sistemas monotarea son aquellos que sólo permiten una tarea a la vez por usuario. Puede darse el caso de un sistema multiusuario y monotarea, en el cual se admiten varios usuarios al mismo tiempo pero cada uno de ellos puede estar haciendo solo una tarea a la vez.

Multitareas: Un sistema operativo multitarea es aquél que le permite al usuario estar realizando varias labores al mismo tiempo. Por ejemplo, puede estar editando el código fuente de un programa durante su depuración mientras compila otro programa, a la vez que está recibiendo correo electrónico en un proceso en background. Es común encontrar en ellos interfaces gráficas orientadas al uso de menús y el ratón, lo cual permite un rápido intercambio entre las tareas para el usuario, mejorando su productividad.

3.4.3. *Por el número de procesadores*: estos se dividen en

Uniproceso: Un sistema operativo uniproceso es aquél que es capaz de manejar solamente un procesador de la computadora, de

Ing. Fausto Rocchi

manera que si la computadora tuviese más de uno le sería inútil. El ejemplo más típico de este tipo de sistemas es el DOS y MacOS.

Multiproceso: Un sistema operativo multiproceso se refiere al número de procesadores del sistema, que es más de uno y éste es capaz de usarlos todos para distribuir su carga de trabajo. Generalmente estos sistemas trabajan de dos formas: simétrica o asimétricamente. Cuando se trabaja de manera asimétrica, el sistema operativo selecciona a uno de los procesadores el cual jugará el papel de procesador maestro y servirá como pivote para distribuir la carga a los demás procesadores, que reciben el nombre de esclavos. Cuando se trabaja de manera simétrica, los procesos o partes de ellos (threads) son enviados indistintamente a cualquiera de los procesadores disponibles, teniendo, teóricamente, una mejor distribución y equilibrio en la carga de trabajo bajo este esquema.

Se dice que un thread es la parte activa en memoria y corriendo de un proceso, lo cual puede consistir de un área de memoria, un conjunto de registros con valores específicos, la pila y otros valores de contexto. Un aspecto importante a considerar en estos sistemas es la forma de crear aplicaciones para aprovechar los varios procesadores. Existen aplicaciones que fueron hechas para correr en sistemas monoproceso que no toman ninguna ventaja a menos que el sistema operativo o el compilador detecte secciones de código paralelizable, los cuales son ejecutados al mismo tiempo en procesadores diferentes. Por otro lado, el programador puede modificar sus algoritmos y aprovechar por sí mismo esta facilidad, pero esta última opción las más de las veces es costosa en horas hombre y muy tediosa, obligando al programador a ocupar tanto o más tiempo a la paralelización que a elaborar el algoritmo inicial.

3.5. Lenguajes de Programación

En informática, es cualquier lenguaje artificial que puede utilizarse para definir una secuencia de instrucciones para su procesamiento por un ordenador o computadora. Es complicado definir qué es y qué no es un lenguaje de programación. Se asume generalmente que es la traducción de las instrucciones a un código que comprende la computadora y que debe ser completamente sistemática. Normalmente es la computadora la que realiza la traducción.

3.5.1. *Lenguaje de maquina*: Vistos a muy bajo nivel, los microprocesadores procesan exclusivamente señales electrónicas binarias. Dar una instrucción a un microprocesador supone en realidad enviar series de unos y ceros espaciadas en el tiempo de una forma determinada. Esta secuencia de señales se denomina código máquina. El código representa normalmente datos y números e instrucciones para manipularlos.

3.5.2. *Lenguajes de bajo nivel*: Un modo más fácil de comprender el código máquina es dando a cada instrucción un nemónico, como por ejemplo `STORE`, `ADD` o `JUMP`. Esta abstracción da como resultado el ensamblador, un lenguaje de muy bajo nivel que es específico de cada microprocesador.

Los lenguajes de bajo nivel permiten crear programas muy rápidos, pero que son a menudo difíciles de aprender. Más importante es el hecho de que los programas escritos en un bajo nivel son prácticamente específicos para cada procesador. Si se quiere ejecutar el programa en otra máquina con otra tecnología, será necesario rescribir el programa desde el principio.

3.5.3. *Lenguajes de alto nivel*: Por lo general se piensa que los ordenadores son máquinas que realizan tareas de cálculos o procesamiento de textos. La descripción anterior es sólo una forma muy esquemática de ver una computadora. Hay un alto nivel de abstracción entre lo que se pide a la computadora y lo que realmente comprende. Existe también una relación compleja entre los lenguajes de alto nivel y el código máquina.

Los lenguajes de alto nivel son normalmente fáciles de aprender porque están formados por elementos de lenguajes naturales, como el inglés. En BASIC, el lenguaje de alto nivel más conocido,

Ing. Fausto Cecchi

los comandos como:

```
IF (CONTADOR = 10) THEN STOP
```

pueden utilizarse para pedir a la computadora que pare si `CONTADOR` es igual a `10`. Por desgracia para muchas personas esta forma de trabajar es un poco frustrante, dado que a pesar de que las computadoras parecen comprender un lenguaje natural, lo hacen en realidad de una forma rígida y sistemática.

3.6. Intérpretes y compiladores

La traducción de una serie de instrucciones en lenguaje ensamblador (el código fuente) a un código máquina (o código objeto) no es un proceso muy complicado y se realiza normalmente por un programa especial llamado *compilador*. La traducción de un código fuente de alto nivel a un código máquina también se realiza con un compilador, en este caso más complejo, o mediante un intérprete. Un compilador crea una lista de instrucciones de código máquina, el código objeto, basándose en un código fuente. El código objeto resultante es un programa rápido y listo para funcionar, pero que puede hacer que falle el ordenador si no está bien diseñado. Los *intérpretes*, por otro lado, son más lentos que los compiladores ya que no producen un código objeto, sino que recorren el código fuente una línea cada vez. Cada línea se traduce a código máquina y se ejecuta. Cuando la línea se lee por segunda vez, como en el caso de los programas en que se reutilizan partes del código, debe compilarse de nuevo. Aunque este proceso es más lento, es menos susceptible de provocar fallos en la computadora.

3.7. Paradigmas de la Programación

Un paradigma se puede considerar como una colección de características abstractas que categorizan un grupo de lenguajes que son aceptados y utilizados por un grupo de profesionales. Los programas de computadora se utilizan para resolver problemas, y ha habido miles de años de trabajo matemático para tal fin. Los lenguajes de programación están especificados por reglas para formar instrucciones correctas, organizándolas en módulos, someterlas hacia un compilador, el cual traduce el código en lenguaje comprensible para una máquina en particular, y finalmente ejecuta el programa, es decir, someter la entrada hacia la computadora, la cual la transforma en una salida de acuerdo con las instrucciones en el programa.

3.7.1. *Paradigmas Imperativos.* Facilitan cálculos por medio de cambios de estado. Entendemos por estado la condición de una RAM o ALMACENAMIENTO.

3.7.2. *Paradigma de procedimientos.* Está constituido por bloques de programas que forman un archivo plano, donde cada bloque sigue a su predecesor, construyéndose así un programa lineal. Es un ejemplo de este paradigma el lenguaje de programación FORTRAN.

3.7.3. *Paradigma estructurado en bloques.* Está constituido por ámbitos anidados, es decir, los bloques pueden estar anidados dentro de otros bloques y pueden contener sus propias variables. Son ejemplos de este paradigma los lenguajes: ADA, ALGOL60, PASCAL, ALGOL68 y C.

3.7.4. *Paradigma basado en objetos.* Describe a los lenguajes que soportan objetos en interacción, tales como ADA (PAQUETES), MÓDULA (MÓDULOS), SMALLTALK (OBJETOS), y además pueden soportar ABSTRACCIÓN, ENCAPSULACIÓN y POLIMORFISMO.

3.7.5. *Paradigma orientado a objetos.* Describe a los lenguajes que están basados en objetos y además soportan clases de objetos y la herencia de atributos de un objeto padre por parte de sus hijos. Son ejemplos de este paradigma los lenguajes: C++, Object Pascal y Java.

3.7.6. *Paradigma del procesamiento en paralelo.* El término PARALELO implica las posiciones de procesadores múltiples, mientras que CONCURRENTE sugiere que los procesos se están ejecutando en forma simultánea. Este paradigma describe los lenguajes que soportan el uso de procesadores múltiples, la cooperación entre

Ing. Fausto Iocchi

procesos y el potencial para falla parcial, es decir, uno o más procesos pueden fallar sin hacer peligrar todo el proyecto. Son ejemplos de este paradigma los lenguajes: Concurrent C, Pascal S, ADA, OCCAM y C-LINDA.

3.7.7. *Paradigma de la programación lógica.* La programación lógica está basada en un subconjunto del cálculo de predicados, incluyendo instrucciones escritas en formas conocidas como cláusulas de HORN, a través del cálculo de estos predicados que proporcionan axiomas y reglas de inferencia se pueden deducir nuevos hechos a partir de hechos conocidos. Una cláusula de HORN permite que solo un nuevo hecho sea deducido en cualquier instrucción simple. Un sistemas de estas cláusulas permite un método particularmente mecánico de demostración llamado resolución. Un programa basado en lógica se compone de una serie de axiomas o hechos, reglas de inferencia y un teorema o cuestión por demostrarse. La salida es cierta si los hechos soportan o apoyan la cuestión y es falsa en el caso contrario.

3.7.8. *Paradigma funcional.* Los lenguajes puramente funcionales operan solamente a través de funciones. Una función devuelve sólo un valor, dada una lista de parámetros.

3.7.9. *Paradigma del lenguaje de bases de datos.* Las propiedades que distinguen a los lenguajes diseñados para tratar con bases de datos son la persistencia y la administración de cambios. Un sistema de administración de bases de datos incluye un Lenguaje Descriptor de Datos (DDL) y un Lenguaje Manipulador de Datos (DML).

3.8. Criterios del lenguaje

Los criterios del lenguaje son características interrelacionadas usadas para considerar si un lenguaje tiene méritos, es decir, si este es confiable y eficiente.

3.8.1. *Sintaxis*: Hablar de sintaxis es hablar de la forma del lenguaje, hablar de la forma del lenguaje es hablar de una colección de instrucciones formadas al seguir un conjunto de reglas que diferencian los programas válidos de los no válidos. Pero la sintaxis por sí misma no da significado a un lenguaje; meramente define la colección de frases y sentencias que son combinaciones válidas de los caracteres de un lenguaje.

3.8.2. *Semántica*: Describe de manera precisa lo que significa una construcción particular. Cada lenguaje de programación podría interpretar de manera distinta dos instrucciones escritas exactamente igual.

3.8.3. *Comprobabilidad*: La prueba de que un programa funciona correctamente involucra tres pasos que describen su comprobabilidad, estos son: la comprobación de que el programa cumple con la intención del programador, probar que el compilador traduce de manera correcta a código de máquina la sintaxis y la semántica del programa en el lenguaje empleado, y comprobar que la máquina misma funciona correctamente.

3.8.4. *Confiabilidad*: El programa se considera confiable si se comporta como es anunciado y produce los resultados que el usuario espera. Para los lenguajes de programación, la confiabilidad por lo general se refiere a los mecanismos que promueven la escritura, mantenimiento y depuración de programas correctos, y el subsiguiente manejo de excepciones cuando un programa se ejecuta.

3.8.5. *Traducción rápida*: Los programas que son escritos en un lenguaje deben traducirse a un lenguaje que la máquina pueda reconocer y por último a un código de máquina que pueda ejecutarse en realidad, la traducción del código fuente debe ser lo más rápido y optimo posible.

3.8.6. *Código fuente eficiente*: Existe un balance comparativo entre el trabajo que el programador debe hacer y el trabajo que el compilador puede hacer. Lenguajes de muy alto nivel donde los programas manipulan estructuras complejas tales como registros,

Ing. Fausto Locchi

listas, relaciones o conjuntos dependen de compiladores de optimización por eficiencia (ejecutan varios pasos del análisis semántico), en contraparte los lenguajes que ejecutan la gama de los parecidos al C, donde el programador puede trabajar muy cerca de la CPU, pudiera agilizar el proceso de compilación, sin embargo el programador debe tener nociones de la máquina en cuestión.

3.8.7. *Ortogonalidad***:** En un lenguaje las variables aleatorias se consideran ortogonales si son independientes entre sí. Es en este sentido de independencia que las características de un lenguaje se consideran ortogonales. Con esto queremos decir que los componentes son independientes entre sí y que se comportan en la misma manera en cualquier circunstancia.

3.8.8. *Transportabilidad***:** Un lenguaje es transportable si sus programas pueden compilarse y ejecutarse en diferentes máquinas sin tener que rescribir el código fuente.

Ing. Fausto Iocchi

Unidad IV

Objetivo:

Previo repaso, lectura e investigación, y la inducción prestada por el facilitador, el participante definirá los diversos conceptos básicos y necesarios de la programación orientada a objetos.

Contenido:

4. Programación Orientada a Objetos.
   4.1. Conceptos de la POO.
      4.1.1. Objeto.
      4.1.2. Clase.
      4.1.3. Encapsulación.
      4.1.4. Abstracción.
      4.1.5. Polimorfismo.
      4.1.6. Herencia.
   4.2. Definición de Clases y Objetos.

Actividades del participante:

- Realizar lectura previa sobre el tema, utilizando la presente guía y otros medios bibliográficos.
- Interactuar en clase.
- Discutir en grupo.
- Realizar las actividades propuestas en clase.

Evaluación multidireccional:

- Evaluación formativa: a través de las discusiones de los conocimientos e intervenciones en clase.
- Evaluación sumativa: a través de un examen práctico y/o parcial.

Programación Orientada a Objetos

4. Programación Orientada a Objetos

La programación orientada a objetos, ha tomado las mejores ideas de la programación estructurada (¿no sabe lo que es la programación estructurada?... investigue) y los ha combinado con varios conceptos nuevos y potentes que incitan a contemplar las tareas de programación desde un nuevo punto de vista. La programación orientada a objetos, permite descomponer más fácilmente un problema en subgrupos de partes relacionadas del problema. Entonces, utilizando el lenguaje se pueden traducir estos subgrupos a unidades auto contenidas llamadas objetos.

El término Programación Orientada a Objetos (POO), hoy en día ampliamente utilizado, es difícil de definir, ya que no es un concepto nuevo, sino que ha sido el desarrollo de técnicas de programación desde principios de la década de los setenta, aunque sea en la década de los noventa cuando ha aumentado su difusión, uso y popularidad. No obstante, se puede definir POO como una técnica o estilo de programación que utiliza objetos como bloque esencial de construcción.

Un objeto es una unidad que contiene datos y las funciones que operan sobre esos datos. A los elementos de un objeto se les conoce como miembros; las funciones que operan sobre los objetos se denominan métodos y los datos se denominan miembros datos.

4.1. Conceptos de la POO

4.1.1. **Objeto:** es una entidad de programa que consiste en datos y todos aquellos procedimientos que pueden manipular aquellos datos; *el acceso a los datos de un objeto es solamente a través de estos procedimientos*, únicamente estos procedimientos pueden manipular, referenciar y/o modificar estos datos.

4.1.2. **Clase:** Podemos considerar una clase como una colección de objetos que poseen características y operaciones comunes. Una

Ing. Fausto Rocchi

clase contiene toda la información necesaria para crear nuevos objetos.

4.1.3. **Encapsulación:** Es una técnica que permite localizar y ocultar los detalles de un objeto. La encapsulación previene que un objeto sea manipulado por operaciones distintas de las definidas. La encapsulación es como una caja negra que esconde los datos y solamente permite acceder a ellos de forma controlada.

Las principales razones técnicas para la utilización de la encapsulación son:

- Mantener a salvo los detalles de representación, si solamente nos interesa el comportamiento del objeto.
- Modificar y ajustar la representación a mejores soluciones algorítmicas o a nuevas tecnologías de software.

4.1.4. **Abstracción:** En el sentido más general, una abstracción es una representación concisa de una idea o de un objeto complicado. En un sentido más especifico, la abstracción localiza y oculta los detalles de un modelo o diseño para generar y manipular objetos.

4.1.5. **Polimorfismo:** Significa que un nombre se puede utilizar para especificar una clase genérica de acciones, es decir, podemos añadir funciones distintas a un solo nombre (función y/o método) u operador.

4.1.6. **Herencia:** La herencia es un proceso mediante el cual un objeto (hijo) puede adquirir las propiedades de otro objeto (padre).

Como resumen de los conceptos expuestos anteriormente, podemos decir dos cosas. A saber:

1) Los objetos son encapsulaciones de abstracciones en la POO.
2) La unidad de encapsulación en la POO es el objeto.

Ahora, la gran pregunta: *¿Cómo diferenciar una clase de un objeto?…* Una clase es un tipo y un objeto es una instancia de ese tipo. Además, la clase es un concepto estático, es decir, una clase es un elemento reconocible en el texto del programa.

Un objeto es un concepto puramente dinámico, el cual pertenece, no al texto del programa, sino a la memoria de la computadora, donde los objetos ocupan un espacio en tiempo de ejecución una vez que haya sido creado.

4.2. Definición de Clases y Objetos

Para crear un objeto, es preciso definir primero su forma general, es decir, debemos definir la clase a la cual pertenece el objeto a crear, para ello utilizamos la palabra reservada **`class`**.

Definición de la clase: una clase es un tipo definido por el usuario que determina la estructura de los datos y las operaciones asociadas a este tipo.

Las clases son como plantillas o modelos que describen como se construyen ciertos tipos de objetos. Cada vez que se construye un objeto de una clase se crea una instancia de esa clase, por consiguiente los objetos son instancias de clases.

La forma general de la declaración de una clase es:

```
class <nombre_de_la_clase> {
  private <datos y funciones privadas de la clase>
  public <datos y funciones públicas de la clase>
  protected <datos y funciones protegidas de la clase>
}
```

Una clase puede contener partes públicas, protegidas y privadas, por defecto, todos los elementos que se definen en la clase son privados; esto significa que no pueden acceder a ellas ninguna función que no sea miembro de la clase.

Una clase puede tener tantas variables como necesite. Estas pueden ser de

Ing. Fausto Iocchi

cualquier tipo, incluyendo otras clases.

Miembros de la clase private:

Los miembros de la clase **private** tienen el más estricto control de acceso. Sólo la clase misma puede tener acceso a un miembro *private*. En este ejemplo nadie puede usar la clase ya que todo es *private*.

```
class Privada {
  private int var1;
  private void Funcion1();
  private void Funcion2();
}

static void main() {
  // Se crea objeto1 de clase Privada.
  Privada objeto1 = new Privada();

  objeto1.var1 = 23;  // acceso no valido por ser private.
  objeto1.Funcion1(); // acceso no valido por ser private.
  objeto1.Funcion2(); // acceso no valido por ser private.
}
```

Para poder tener acceso necesitaría que las funciones miembro fueran declaradas en la sección *public*.

Miembros de la clase public:

Para utilizar los datos y funciones miembro de una clase en otra que no tenga relación alguna, se deben declarar las funciones y datos en la sección *public*.

```
class Publica {
  public int var1;
  public void Funcion1();
  public void Funcion2();
}

static void main() {
  Publica objeto1 = new Publica(); // Se crea objeto1: clase Publica.

  objeto1.var1 = 23;               // acceso válido.
  objeto1.Funcion1();              // acceso válido.
  objeto1.Funcion2();              // acceso válido.
}
```

Cualquier miembro que se declare en la sección *public*, hace posible el acceso ilimitado de dicho miembro desde cualquier clase y/o método fuera de la clase que la declare.

Miembros de la clase protected:

Cuando se define una clase que se utiliza subsiguientemente como clase de base para otras clases (clase padre), se puede hacer que los miembros estén accesibles solo para funciones de las clases derivadas mediante el uso de la palabra clave *protected*.

Considere las clases siguientes, según su jerarquía:

```
class Animales {
  protected int Npatas;
  protected int Nojos;
  protected void desplazarse();
}

class Perro: Animales {
  protected string raza;
  protected void Asignar_Patas() {
    // acceso al campo Npatas de Animales válido por ser protected.
    this.Npatas = 4;
  }
}

static void main() {
  Perro P1 = new Perro();     // Se crea P1 de clase Perro.
  P1.raza = "Pastor";         // acceso no válido, porque la función
                              // main() no tiene relación con la
                              // clase Perro.
}
```

Los campos protegidos sólo pueden ser accedidos por las funciones miembros de esa clase y las funciones miembro de las clases derivadas (clases hijas).

Constructores de una clase:

Un constructor es una función especial que es miembro de esa clase y que tiene el mismo nombre de la clase. Es muy frecuente que una cierta parte de un objeto necesite una iniciación antes de que pueda ser utilizada;

Ing. Fausto Cocchi

como el requisito de iniciación es tan frecuente, C# permite que los objetos se den a sí mismos valores iniciales cuando se crean. Esta iniciación automáticamente se lleva a cabo mediante el uso de una función de construcción o constructor.

Por ejemplo, si tenemos una clase que simule el tanque de gasolina de un carro el constructor debería ser algo como el código que se muestra a continuación. En dicho código se coloca el modificador **this** para hacer mención a un miembro (bien sea campo o función), de la clase actual. Como se podrá observar, al principio de la creación del objeto (en el constructor), suponemos que el tanque del carro está completamente lleno.

```
class Tanque {
  protected int TipoGasolina;
  protected int CantidadGasolina;
  protected void Tanque(int TipoGas) {
    this.TipoGasolina = TipoGas;
    this.CantGasolina = 100;      // 100 es la máxima cantidad
  }
}
```

La función de construcción de un objeto se invoca cuando se crea el objeto. Esto significa que se invoca cuando se ejecuta la declaración del objeto. Como lo dice el nombre, un constructor es una función que se utiliza para construir un objeto de una clase dada; esto puede implicar la presencia de diferentes escenarios.

1) Creación de objetos con iniciación definida.
2) Creación de objetos con iniciación especifica.
3) Creación de objetos copiando otro objeto.

Cada uno de estos procesos implica un tipo diferente de constructor. Un constructor tiene el nombre de la clase a la que pertenece.

Destructores de una clase:

Los destructores entran en la misma categoría que los constructores. Se utilizan para realizar ciertas operaciones que son necesarias cuando ya no se utiliza un objeto como es la liberación de memoria.

Existen algunas diferencias importantes entre los constructores y los destructores:

1) Los destructores pueden ser virtuales, los constructores NO.
2) A los destructores no se les puede mandar argumentos.
3) Sólo se puede declarar un destructor para una clase dada.

El destructor se nombra como la clase (o sea, con el mismo nombre), pero este va precedido de una tilde (~).

```
class Tanque
{
  protected int TipoGasolina;
  protected int CantidadGasolina;

  protected void ~Tanque()
  {
    this.CantGasolina = 0;
  }
}
```

El destructor se utiliza, generalmente, para desechar cualquier espacio de memoria asignado al objeto.

Ing. Fausto Iocchi

Unidad V

Objetivo:

Previo repaso de la unidad IV y la lectura e investigación, aunado con la inducción prestada por el facilitador, el participante empleará su conocimiento para realizar programas de computación básicos, en los que resolverá problemas variados.

Contenido:

5. Aspectos Léxicos.
   5.1. Comentarios.
   5.2. Literales.
   5.3. Operadores.
   5.4. Variables y tipos de datos.
   5.5. Instrucciones.
      5.5.1. Instrucciones básicas.
      5.5.2. Instrucciones condicionales.
      5.5.3. Instrucciones iterativas.
      5.5.4. Instrucciones de salto.

Actividades del participante:

- Realizar lectura previa sobre el tema, utilizando la presente guía y otros medios bibliográficos.
- Interactuar en clase.
- Discutir en grupo.
- Realizar las actividades propuestas en clase.

Evaluación multidireccional:

- Evaluación formativa: a través de las discusiones de los conocimientos e intervenciones en clase.
- Evaluación sumativa: a través de un examen práctico y/o parcial.

Datos Simples y Estructuras de Control

5. Aspectos Léxicos

Una vez adquirida la lógica de programación, es necesario conocer un software de desarrollo (lenguaje de programación), el cual utilizaremos para transcribir de forma sistemática el software que se desea crear. Para ello debemos saber cómo funciona el lenguaje a utilizar, es decir, conocer sus aspectos léxicos, como declarar sus instrucciones, asignaciones, ciclos, funciones, etc.

Como ya se dijo en algún momento, el objetivo de esta guía está orientada a la enseñanza de los conceptos de la POO, y para esto nos apoyaremos en un lenguaje de programación denominado C#, propio del paquete de programación Microsoft Visual Studio .NET.

5.1. Comentarios

Un comentario es texto que se incluye en el código fuente para facilitar su lectura a los programadores y cuyo contenido es, por defecto, completamente ignorado por el compilador. Suelen usarse para incluir información sobre el autor del código, para aclarar el significado o el porqué de determinadas secciones de código, para describir el funcionamiento de los métodos de las clases, etc.

En C# hay dos formas de escribir comentarios. La primera consiste en encerrar todo el texto que se desee comentar entre caracteres **/*** y ***/** siguiendo la siguiente sintaxis:

```
/* <texto> */
```

Estos comentarios pueden abarcar tantas líneas como sea necesario. Por ejemplo:

```
/* Esto es un comentario
   que ejemplifica cómo se escribe comentarios que
   ocupen varias líneas */
```

Ing. Fausto Iocchi

Ahora bien, hay que tener cuidado con el hecho de que no es posible anidar comentarios de este tipo. Es decir, no vale escribir comentarios como el siguiente:

```
/* Comentario contenedor /* Comentario contenido */ */
```

Esto se debe a que como el compilador ignora todo el texto contenido en un comentario y sólo busca la secuencia `*/` que marca su final, ignorará el segundo `/*` y cuando llegue al primer `*/` considerará que ha acabado el comentario abierto con el primer `/*` (no el abierto con el segundo) y pasará a buscar código. Como el `*/` sólo lo admite si ha detectado antes algún comentario abierto y aún no cerrado (no mientras busca código), cuando llegue al segundo `*/` considerará que ha habido un error ya que encontrará el `*/` donde esperaba encontrar código.

Dado que muchas veces los comentarios que se escriben son muy cortos y no suelen ocupar más de una línea, C# ofrece una sintaxis alternativa más compacta para la escritura este tipo de comentarios en las que se considera como indicador del comienzo del comentario la pareja de caracteres `//` y como indicador de su final el fin de línea. Por tanto, la sintaxis que siguen estos comentarios es:

```
// <texto>
```

Y un ejemplo de su uso es:

```
// Acá se muestra cómo escribir comentarios abreviados de una línea
```

Estos comentarios de una sola línea sí que pueden anidarse sin ningún problema. Por ejemplo, el siguiente comentario es perfectamente válido:

```
// Comentario contenedor // Comentario contenido
```

5.2. Literales

Un literal es la representación explícita de los valores que pueden tomar los tipos básicos del lenguaje. A continuación se explica cuál es la sintaxis con que se escriben los literales en C# desglosándolos según el tipo de valores que representan:

- *Literales enteros*: Un número entero se puede representar en C# tanto en formato decimal como hexadecimal. En el primer caso basta escribir los dígitos decimales (0-9) del número unos tras otros, mientras que en el segundo hay que preceder los dígitos hexadecimales (0-9, a-f, A-F) con el prefijo **0x**. En ambos casos es posible preceder el número de los operadores **+** ó **-** para indicar si es positivo o negativo, aunque si no se pone nada se considerará que es positivo. Ejemplos de literales enteros son 0, 5, +15, -23, 0x1A, -0x1a, etc.
- *Literales de cadena*: Una **cadena** no es más que una secuencia de caracteres encerrados entre comillas dobles. Por ejemplo `"Hola, mundo"`, `"camión"`, etc. El texto contenido dentro estos literales puede estar formado por cualquier número de literales de carácter concatenados y sin las comillas simples, aunque si incluye comillas dobles éstas han de escribirse usando secuencias de escape porque si no el compilador las interpretaría como el final de la cadena.
- *Literal nulo*: El **literal nulo** es un valor especial que se representa en C# con la palabra reservada **`null`** y se usa como valor de las variables de objeto no inicializadas para así indicar que contienen referencias nulas.

5.3. Operadores

Un operador es un símbolo formado por uno o más caracteres, que permite realizar una determinada operación entre uno o más datos y produce un resultado, que bien podremos, por ejemplo, asignárselo a alguna variable declarada previamente.

A continuación se describen cuáles son los operadores incluidos en el lenguaje, clasificados según el tipo de operaciones que permiten realizar, aunque hay que tener en cuenta que C# permite la redefinición del significado de la mayoría de los operadores, según el tipo de dato sobre el que se apliquen, por lo que lo que aquí se cuenta se corresponde con los usos más comunes de los mismos:

Ing. Fausto Cecchi

- *Operaciones aritméticas*: Los operadores aritméticos son los típicos de suma (**+**), resta (**-**), producto (*****), división (**/**) y módulo (**%**).
- *Operaciones lógicas*: Se incluyen operadores que permiten realizar las operaciones lógicas típicas: `and` (**&&** y **&**), `or` (**||** y **|**), `not` (**!**).

Los operadores **&&** y **||** se diferencian de **&** y **|** en que los primeros realizan evaluación perezosa y los segundos no. La evaluación perezosa consiste en que si el resultado de evaluar el primer operando permite deducir el resultado de la operación, entonces no se evalúa el segundo y se devuelve dicho resultado directamente, mientras que la evaluación no perezosa consiste en evaluar siempre ambos operandos. Es decir, si el primer operando de una operación **&&** es falso se devuelve **`false`** directamente, sin evaluar el segundo; y si el primer operando de una **||** es cierto se devuelve **`true`** directamente, sin evaluar el otro.

- *Operaciones relacionales*: Se han incluido los tradicionales operadores de igualdad (**==**), desigualdad (**!=**), mayor que (**>**), menor que (**<**), mayor o igual que (**>=**) y menor o igual que (<=).
- *Operaciones de asignación*: Para realizar asignaciones se usa el operador **=**, operador que además de realizar la asignación que se le solicita devuelve el valor asignado. Por ejemplo, la expresión **`a = b`** asigna a la variable `a` el valor de la variable `b` y devuelve dicho valor, mientras que la expresión `c = a = b` asigna a las variables `c` y `a` el valor de `b` (el operador **=** es asociativo por la derecha).

También se han incluido operadores de asignación compuestos que permiten ahorrar tecleo a la hora de realizar asignaciones tan comunes como:

```
temperatura = temperatura + 15;  // Sin usar asignación compuesta
temperatura += 15;               // Usando asignación compuesta
```

Las dos líneas anteriores son equivalentes, pues el operador compuesto **+=** asigna a su primer operando el valor que tenía más el de su segundo operando (o sea, le suma el segundo operando). Como se ve, permite compactar bastante el código.

Aparte del operador de asignación compuesto **+=**, también se ofrecen operadores de asignación compuestos para la mayoría de los operadores binarios ya vistos. Estos son: **+=**, **-=**, ***=**, **/=**, **%=**.

Otros dos operadores de asignación incluidos son los de incremento (**++**) y decremento (**--**). Estos operadores permiten, respectivamente, aumentar y disminuir en una unidad el valor de la variable sobre el que se aplican. Así, estas líneas de código son equivalentes:

```
temperatura = temperatura + 1; // Sin usar asignación compuesta
temperatura += 1;              // Usando asignación compuesta
temperatura++;                 // Usando incremento
```

Si el operador **++** se coloca tras el nombre de la variable (como en el ejemplo anterior), devuelve el valor de la variable **antes** de incrementarla, mientras que si se coloca delante, devuelve el valor de ésta **después** de incrementarla; y lo mismo ocurre con el operador **--**. Por ejemplo:

```
c = b++; // Se asigna a c el valor de b y luego se incrementa b
c = ++b; // Se incrementa el valor de b y luego se asigna a c
```

La ventaja de usar los operadores **++** y **--** es que en muchas máquinas son más eficientes que el resto de formas de realizar sumas o restas de una unidad, pues el compilador puede traducirlos en una única instrucción en código máquina.

- *Operaciones con cadenas*: Para realizar operaciones de concatenación de cadenas se puede usar el mismo operador que para realizar sumas, ya que en C# se ha redefinido su significado para que cuando se aplique entre operandos que sean cadenas o que sean una cadena y un carácter lo que haga sea concatenarlos. Por ejemplo, `"Hola" + " mundo"` devuelve `"Hola mundo"`, y `"Hola mund" + 'o'` también.
- *Operaciones de acceso a arreglos*: Un **arreglo** es un conjunto ordenado de objetos de tamaño fijo. Para acceder a cualquier elemento de este conjunto se aplica el operador postfijo **[]** sobre el arreglo para indicar entre corchetes la posición que ocupa el objeto al que se desea acceder

Ing. Fausto Jocchi

dentro del conjunto. Es decir, este operador se usa así:

```
[<posiciónElemento>]
```

Un ejemplo de su uso en el que se asigna al elemento que ocupa la posición 3 en un arreglo de nombre *tablaPrueba* el valor del elemento que ocupa la posición 18 de dicha tabla es el siguiente:

```
tablaPrueba[3] = tablaPrueba[18];
```

Los arreglos se estudiarán detenidamente en la siguiente unidad: *Datos Compuestos*, por lo que si no entiende esto ahora, no se preocupe, solo siga leyendo.

- *Operaciones de acceso a objetos*: Para acceder a los miembros de un objeto se usa el operador **.**, cuya sintaxis es:

```
<objeto>.<miembro>
```

Si a es un objeto, ejemplos de cómo llamar a diferentes miembros suyos son:

```
a.b = 2;  // Asignamos a su propiedad a el valor 2
a.f();    // Llamamos a su método f()
a.g(2);   // Llamamos a su método g() pasándole como parámetro
          // el valor entero 2
```

- *Operaciones de creación de objetos*: El operador más típicamente usado para crear objetos es **new**, que se usa así:

```
new <nombreTipo>(<parametros>)
```

Este operador crea un objeto de `<nombreTipo>` pasándole a su método constructor los parámetros indicados en `<parámetros>` y devuelve una referencia al mismo. En función del tipo y número de estos parámetros se llamará a uno u otro de los constructores del objeto. Así, suponiendo que `a1` y `a2` sean variables de tipo `Avión`, ejemplos de uso del operador **new** son:

```
// Se llama al constructor sin parámetros de Avión
Avión a1 = new Avión();

// Se llama al constructor de Avión que toma como
// parámetro una cadena
Avión a2 = new Avión("Caza");
```

- *Operaciones de conversión (cast)*: Para convertir unos objetos en otros se utiliza el operador de conversión, que no consiste más que en preceder la expresión a convertir del nombre entre paréntesis del tipo al que se desea convertir el resultado de evaluarla. Por ejemplo, si L es una variable de tipo **long** y se desea almacenar su valor dentro de una variable de tipo **int** llamada `i`, habría que convertir previamente su valor a tipo **int** así:

```
int i;
long L = 1;

// Asignamos a i el resultado de convertir el valor de L a tipo int
i = (int) L;
```

Los tipos **int** y **long** están predefinidos en C# y permite almacenar valores enteros con signo. La capacidad de **int** es de 32 bits, mientras que la de **long** es de 64 bits. Por tanto, a no ser que hagamos uso del operador de conversión, el compilador no nos dejará hacer la asignación, ya que al ser mayor la capacidad de los **long**, no todo valor que se pueda almacenar en un **long** tiene porqué poderse almacenar en un **int**. Es decir, no es válido:

```
int i;
long L = 1;

i = L; //ERROR: El valor de L no tiene porqué caber en i
```

Esta restricción en la asignación la impone el compilador debido a que sin ella podrían producirse errores muy difíciles de detectar ante truncamientos no esperados debido al que el valor de la variable fuente es superior a la capacidad de la variable destino.

Nota: Para poder convertir una cadena en número es necesario colocar

Ing. Fausto Rocchi

la función `Int32.Parse(<texto>)`, y colocar la cadena numérica en el campo `<texto>`. Ejemplo:

```
X = Int32.Parse("334");     // X es de tipo numérico
```

5.4. Variables y tipos de datos

Una variable puede verse simplemente como un hueco en el que se puede almacenar un objeto de un determinado tipo, al que se le da un cierto nombre. Para poderla utilizar sólo hay que definirla indicando cual será su nombre y cuál será el tipo de datos que podrá almacenar, lo que se hace siguiendo la sintaxis:

```
<tipoVariable> <nombreVariable>;
```

Una variable puede ser definida dentro de una definición de clase, en cuyo caso se correspondería con el tipo de miembro que hasta ahora hemos denominado **campo** (y conocido en POO como **atributo**).

También puede definirse como una **variable local** a un método, que es una variable definida dentro del código del método a la que sólo puede accederse desde dentro de dicho código. Otra posibilidad es definirla como **parámetro** de un método, que son variables que almacenan los valores de llamada al método y que, al igual que las variables locales, sólo pueden ser accedidas desde código ubicado dentro del método. El siguiente ejemplo muestra cómo definir variables de todos estos casos:

```
class A
{
  int x, z;                        // Son los campos
  int y;

  void F(string a, string b) {     // Son los parámetros
    Persona p;                     // Es la variable local
  }
}
```

En este ejemplo las variables `x`, `z` e `y` son campos de tipo **`int`**, mientras que **`p`** es una variable local de tipo `Persona` y `a` y `b` son parámetros de tipo **`string`**. Como se muestra en el ejemplo, si un método toma varios parámetros las definiciones de éstos se separan mediante comas (carácter ','), y si queremos definir varios campos o variables locales (no válido para parámetros) de un mismo tipo podemos incluirlos en una misma definición incluyendo en `<nombreVariable>` sus nombres separados por comas.

Con la sintaxis de definición de variables anteriormente dada simplemente definimos variables pero no almacenamos ningún objeto inicial en ellas.

Ya hemos visto que para crear objetos se utiliza el operador **`new`**. Por tanto, una forma de asignar un valor a la variable `p` del ejemplo anterior sería así:

```
Persona p;

p = new Persona("José", 22, "dirección # 1");
```

Sin embargo, C# también proporciona una sintaxis más sencilla con la que podremos asignar un objeto a una variable en el mismo momento que se define. Para ello se la ha de definir usando esta otra notación:

```
<tipoVariable> <nombreVariable> = <valorInicial>;
```

Así por ejemplo, la anterior asignación de valor a la variable `p` podría reescribirse de esta otra forma más compacta:

```
Persona p = new Persona("José", 22, "otra dirección");
int x = 3;
```

La especificación de un valor inicial también combinarse con la definición de múltiples variables separadas por comas en una misma línea. Por ejemplo, las siguientes definiciones son válidas:

```
Persona p1 = new Persona("José", 22, 76543876),
        p2 = new Persona("Juan", 21, 87654212);
int x = 3, y = 2;
```

Ing. Fausto Iocchi

Y son tratadas por el compilador de forma completamente equivalentes a haberlas declarado como:

```
Persona p1 = new Persona("José", 22, 76543876);
Persona p2 = new Persona("Juan", 21, 87654212);
int x = 3;
int y = 2;
```

Los tipos de datos básicos son ciertos tipos de datos tan comúnmente utilizados en la escritura de aplicaciones que en C# se ha incluido una sintaxis especial para tratarlos. Por ejemplo, para representar números enteros de 32 bits con signo se utiliza el tipo de dato **`System.Int32`** definido en la BCL (librería propia del lenguaje C#), aunque a la hora de crear un objeto a de este tipo que represente el valor 2 se usa la siguiente sintaxis:

```
System.Int32 a = 2;
```

Como se ve, no se utiliza el operador **`new`** para crear objeto **`System.Int32`**, sino que directamente se indica el literal que representa el valor a crear, con lo que la sintaxis necesaria para crear entero de este tipo se reduce considerablemente. Es más, dado lo frecuente que es el uso de este tipo también se ha predefinido en C# el alias **`int`** para el mismo, por lo que la definición de variable anterior queda así de compacta:

```
int a = 2;
```

Nota: El valor que por defecto se da a los campos de tipos básicos consiste en poner a cero toda el área de memoria que ocupen. Esto se traduce en que los campos de tipos básicos numéricos se inicializan por defecto con el valor 0, los de tipo **bool** lo hacen con **false**, los de tipo **char** con '\u0000', y los de tipo **string** y **object** con **null**.

`System.Int32` no es el único tipo de dato básico incluido en C#. En el espacio de nombres **`System`** se han incluido todos los que a continuación se muestran en la siguiente tabla.

Tipo	Descripción	Bits	Rango de valores	Alias
SByte	*Bytes* con signo	8	[-128, 127]	sbyte
Byte	*Bytes* sin signo	8	[0, 255]	byte
Int16	Enteros cortos con signo	16	[-32.768, 32.767]	short
UInt16	Enteros cortos sin signo	16	[0, 65.535]	ushort
Int32	Enteros normales	32	[-2.147.483.648, 2.147.483.647]	int
UInt32	Enteros normales sin signo	32	[0, 4.294.967.295]	uint
Int64	Enteros largos	64	[-9.223.372.036.854.775.808, 9.223.372.036.854.775.807]	long
UInt64	Enteros largos sin signo	64	[0-18.446.744.073.709.551.615]	ulong
Single	Reales con 7 dígitos de precisión	32	[$1{,}5\times10^{-45}$ - $3{,}4\times10^{38}$]	float
Double	Reales de 15-16 dígitos de precisión	64	[$5{,}0\times10^{-324}$ - $1{,}7\times10^{308}$]	double
Decimal	Reales de 28-29 dígitos de precisión	128	[$1{,}0\times10^{-28}$ - $7{,}9\times10^{28}$]	decimal
Boolean	Valores lógicos	32	**true**, **false**	bool
Char	Caracteres Unicode	16	['\u0000', '\uFFFF']	char
String	Cadenas de caracteres	Variable	El permitido por la memoria	string
Object	Cualquier objeto	Variable	Cualquier objeto	object

Tipos de datos básicos

5.5. Instrucciones

Una instrucción es toda acción que se pueda realizar en el cuerpo de un método, como definir variables locales, llamar a métodos, asignaciones y muchas cosas más que veremos a lo largo de este tema.

Las instrucciones se agrupan formando **bloques de instrucciones**, que son listas de instrucciones encerradas entre llaves que se ejecutan una tras otra. Es decir, la sintaxis que se sigue para definir un bloque de instrucciones es:

Ing. Faustolocchi

```
{
  <listaInstrucciones>
}
```

Toda variable que se defina dentro de un bloque de instrucciones sólo existirá dentro de dicho bloque. Tras él será inaccesible y podrá ser destruida por el recolector de basura. Por ejemplo, este código no es válido:

```
public void f()
{
  {
    int b;
  }
  b = 1;     // ERROR: b no existe fuera del bloque donde se declaró
}
```

Los bloques de instrucciones pueden anidarse, aunque si dentro de un bloque interno definimos una variable con el mismo nombre que otra definida en un bloque externo se considerará que se ha producido un error, ya que no se podrá determinar a cuál de las dos se estará haciendo referencia cada vez que se utilice su nombre en el bloque interno.

5.5.1. Instrucciones básicas

Anteriormente se vio que las **variables locales** son variables que se definen en el cuerpo de los métodos y sólo son accesibles desde dichos cuerpos. Recuérdese que la sintaxis explicada para definirlas era la siguiente:

```
<modificadores> <tipoVariable> <nombreVariable> = <valor>;
```

También ya entonces se vio que podían definirse varias variables en una misma instrucción separando sus pares nombre-valor mediante comas, como por ejemplo:

```
int a = 5, b, c = -1;
```

- *Asignaciones*: Una asignación es simplemente una instrucción mediante la que se indica un valor a almacenar en un dato. La sintaxis usada para ello es:

```
<destino> = <origen>;
```

En temas previos ya se han dado numerosos ejemplos de cómo hacer esto, por lo que no es necesario hacer ahora mayor hincapié en ello.

- *Llamadas a métodos*: Una llamada a un método consiste en solicitar la ejecución de sus instrucciones asociadas dando a sus parámetros ciertos valores. Si el método a llamar es un método de objeto, la sintaxis usada para ello es:

```
<objeto>.<nombreMétodo>(<valoresParámetros>);
```

Y si el método a llamar es un método de tipo, entonces la llamada se realiza con:

```
<nombreTipo>.<nombreMétodo>(<valoresParámetros>);
```

5.5.2. Instrucciones condicionales

Las instrucciones condicionales son instrucciones que permiten ejecutar bloques de instrucciones sólo si se da una determinada condición. En los siguientes subapartados de este epígrafe se describen cuáles son las instrucciones condicionales disponibles en C#.

- *Instrucción if*: La instrucción **if** permite ejecutar ciertas instrucciones sólo si se da una determinada condición. Su sintaxis de uso es la sintaxis:

```
if (<condición>)
{
  <instruccionesIf>
}
else
{
  <instruccionesElse>
}
```

Ing. Fausto Iocchi

El significado de esta instrucción es el siguiente: se evalúa la expresión <condición>, que ha de devolver un valor lógico. Si es cierta (devuelve **true**) se ejecutan las <instruccionesIf>, y si es falsa (**false**) se ejecutan las <instruccionesElse>. La rama *else* **es opcional**, y si se omite y la condición es falsa se seguiría ejecutando a partir de la instrucción siguiente al **if**. En realidad, tanto <instruccionesIf> como <instruccionesElse> pueden ser una única instrucción o un bloque de instrucciones.

Un ejemplo de aplicación de esta instrucción es esta variante del HolaMundo:

```
using System;

class HolaMundoIf {
  public static void Main(String[] args) {
    if (args.Length > 0) {
      Console.WriteLine("Hola {0}!", args[0]);
    } else {
      Console.WriteLine("Hola mundo!");
    }
  }
}
```

Si ejecutamos este programa sin ningún argumento veremos que el mensaje que se muestra es `¡Hola Mundo!`, mientras que si lo ejecutamos con algún argumento se mostrará un mensaje de bienvenida personalizado con el primer argumento indicado.

- *Instrucción switch*: La instrucción **switch** permite ejecutar unos u otros bloques de instrucciones según el valor de una cierta expresión. Su estructura es:

```
switch (<expresión>)
{
  case <valor1>: <bloque1>
                 <siguienteAcción>
  case <valor2>: <bloque2>
                 <siguienteAcción>
  ...
  default: <bloqueDefault>
           <siguienteAcción>
}
```

El significado de esta instrucción es el siguiente: se evalúa <expresión>. Si su valor es <valor1> se ejecuta el <bloque1>, si es <valor2> se ejecuta <bloque2>, y así para el resto de valores especificados. Si no es igual a ninguno de esos valores y se incluye la rama **default**, se ejecuta el <bloqueDefault>; pero si no se incluye se pasa directamente a ejecutar la instrucción siguiente al **switch**.

Los valores indicados en cada rama del **switch** han de ser expresiones constantes que produzcan valores de algún tipo básico entero, de una enumeración, de tipo **char** o de tipo **string**. Además, no puede haber más de una rama con el mismo valor.

En realidad, aunque todas las ramas de un **switch** son opcionales siempre se ha de incluir al menos una. Además, la rama **default** no tiene porqué aparecer la última si se usa, aunque es recomendable que lo haga para facilitar la legibilidad del código.

El elemento marcado como <siguienteAcción> colocado tras cada bloque de instrucciones indica qué es lo que ha de hacerse tras ejecutar las instrucciones del bloque que lo preceden. Puede ser uno de estos tres tipos de instrucciones:

```
goto case <valori>;
goto default;
break;
```

Si es un **goto case** indica que se ha de seguir ejecutando el bloque de instrucciones asociado en el **switch** a la rama del <valori> indicado, si es un **goto default** indica que se ha de seguir ejecutando el bloque de instrucciones de la rama **default**, y si es un **break** indica que se ha de seguir ejecutando la instrucción siguiente al switch.

El siguiente ejemplo muestra cómo se utiliza **switch**. Este programa lo que hace es reconocer ciertos nombres de personas que se le pueden pasar como argumento al ejecutarlo y le saluda de forma especial. La rama **default** se incluye para dar un saludo por defecto a las personas no reconocidas.

Ing. Fausto Locchi

```
using System;

class HolaMundoSwitch {
  public static void Main(String[] args) {
    if (args.Length > 0)
      switch (args[0]) {
        case "José": Console.WriteLine("Hola José. Buenos días");
                     break;
        case "Paco": Console.WriteLine("Hola Paco.");
                     break;
        default: Console.WriteLine("Hola {0}", args[0]);
                 break;
      } else
        Console.WriteLine("Hola Mundo");
  }
}
```

Para los programadores habituados a lenguajes como C++ es importante resaltarles el hecho de que, a diferencia de dichos lenguajes, C# obliga a incluir una sentencia **break** o una sentencia **goto case** al final de cada rama del **switch** para evitar errores comunes y difíciles de detectar causados por olvidar incluir **break;** al final de alguno de estos bloques y ello provocar que tras ejecutarse ese bloque se ejecute también el siguiente.

5.5.3. Instrucciones iterativas

Las instrucciones iterativas son instrucciones que permiten ejecutar repetidas veces una instrucción o un bloque de instrucciones mientras se cumpla una condición. Es decir, permiten definir bucles donde ciertas instrucciones se ejecuten varias veces. A continuación se describen cuáles son las instrucciones de este tipo incluidas en C#.

- *Instrucción while*: La instrucción **while** permite ejecutar un bloque de instrucciones mientras se dé una cierta instrucción. Su sintaxis es:

```
while (<condición>)
{
  <instrucciones>
}
```

Su significado es el siguiente: Se evalúa la <condición> indicada, que ha de producir un valor lógico. Si es cierta (valor lógico **true**) se ejecutan las <instrucciones> y se repite el proceso de evaluación de <condición> y ejecución de <instrucciones> hasta que deje de serlo. Cuando sea falsa (**false**) se pasará a ejecutar la instrucción siguiente al **while**. En realidad <instrucciones> puede ser una única instrucción o un bloque de instrucciones. Un ejemplo cómo utilizar esta instrucción es el siguiente:

```
using System;

class HolaMundoWhile
{
  public static void Main(String[] args)
  {
    int actual = 0;

    if (args.Length > 0)
      while (actual < args.Length)
      {
        Console.WriteLine("¡Hola {0}!", args[actual]);
        actual = actual + 1;
      }
    else
      Console.WriteLine("¡Hola mundo!");
  }
}
```

En este caso, si se indica más de un argumento al llamar al programa se mostrará por pantalla un mensaje de saludo para cada uno de ellos. Para ello se usa una variable actual que almacena cuál es el número de argumento a mostrar en cada ejecución del **while**. Para mantenerla siempre actualizada lo que se hace es aumentar en una unidad su valor tras cada ejecución de las <instrucciones> del bucle.

Por otro lado, dentro de las <instrucciones> de un **while** pueden utilizarse las siguientes dos instrucciones especiales:

- **break;**: Indica que se ha de abortar la ejecución del bucle y continuarse ejecutando por la instrucción siguiente al **while**.
- **continue;**: Indica que se ha de abortar la ejecución de las <instrucciones> y reevaluarse la <condición> del bucle, volviéndose

Ing. Fausto Iocchi

a ejecutar las <instrucciones> si es cierta o pasándose a ejecutar la instrucción siguiente al **while** si es falsa.

- *Instrucción do…while*: La instrucción **do...while** es una variante del **while** que se usa así:

```
do
{
  <instrucciones>
} while(<condición>);
```

La única diferencia del significado de **do...while** respecto al de **while** es que en vez de evaluar primero la condición y ejecutar <instrucciones> sólo si es cierta, **do...while** primero ejecuta las <instrucciones> y luego mira la <condición> para ver si se ha de repetir la ejecución de las mismas. Por lo demás ambas instrucciones son iguales, e incluso también puede incluirse **break;** y **continue;** entre las <instrucciones> del **do...while**.

do ... while está especialmente destinado para los casos en los que haya que ejecutar las <instrucciones> al menos una vez aún cuando la condición sea falsa desde el principio, como ocurre en el siguiente ejemplo:

```
using System;

class HolaMundoDoWhile
{
  public static void Main()
  {
    string leído;

    do
    {
      Console.WriteLine("Clave: ");
      leído = Console.ReadLine();
    } while (leído != "José");

    Console.WriteLine("Hola José");
  }
}
```

Este programa pregunta al usuario una clave y mientras no introduzca la correcta (José) no continuará ejecutándose. Una vez que introducida correctamente dará un mensaje de bienvenida al usuario.

- *Instrucción for*: La instrucción **for** es una variante de **while** que permite reducir el código necesario para escribir los tipos de bucles más comúnmente usados en programación. Su sintaxis es:

```
for (<inicialización>; <condición>; <modificación>)
{
  <instrucciones>
}
```

El significado de esta instrucción es el siguiente: se ejecutan las instrucciones de <inicialización>, que suelen usarse para definir e inicializar variables que luego se usarán en <instrucciones>. Luego se evalúa <condición>, y si es falsa se continúa ejecutando por la instrucción siguiente al **for**; mientras que si es cierta se ejecutan las <instrucciones> indicadas, luego se ejecutan las instrucciones de <modificación> -que como su nombre indica suelen usarse para modificar los valores de variables que se usen en <instrucciones>- y luego se reevalúa <condición> repitiéndose el proceso hasta que ésta última deje de ser cierta.

En <inicialización> puede en realidad incluirse cualquier número de instrucciones que no tienen porqué ser relativas a inicializar variables o modificarlas, aunque lo anterior sea su uso más habitual. En caso de ser varias se han de separar mediante comas (**,**), ya que el carácter de punto y coma (**;**) habitualmente usado para estos menesteres se usa en el **for** para separar los bloques de <inicialización>, <condición> y <modificación> Además, la instrucción nula no se puede usar en este caso y tampoco pueden combinarse definiciones de variables con instrucciones de otros tipos.

Con <modificación> pasa algo similar, ya que puede incluirse código que nada tenga que ver con modificaciones pero en este caso no se pueden incluir definiciones de variables.

Como en el resto de instrucciones hasta ahora vistas, en <instrucciones>

Ing. Fausto Jocchi

puede ser tanto una única instrucción como un bloque de instrucciones. Además, las variables que se definan en <inicialización> serán visibles sólo dentro de esas <instrucciones>.

La siguiente clase es equivalente a la clase HolaMundoWhile ya vista solo que hace uso del **for** para compactar más su código:

```
using System;

class HolaMundoFor {
  public static void Main(String[] args) {
    if (args.Length > 0)
      for (int actual = 0; actual < args.Length; actual++)
        Console.WriteLine("¡Hola {0}!", args[actual]);
    else
      Console.WriteLine("¡Hola mundo!");
  }
}
```

Al igual que con **while**, dentro de las <instrucciones> del **for** también pueden incluirse instrucciones **continue;** y **break;** que puedan alterar el funcionamiento normal del bucle.

5.5.4. Instrucciones de salto

Las instrucciones de salto permiten variar el orden normal en que se ejecutan las instrucciones de un programa, que consiste en ejecutarlas una tras otra en el mismo orden en que se hubiesen escrito en el código fuente. Posteriormente se describirán cuáles son las instrucciones de salto incluidas en C#:

- *Instrucción break*: Ya se ha visto que la instrucción **break** sólo puede incluirse dentro de bloques de instrucciones asociados a instrucciones iterativas o instrucciones **switch** e indica que se desea abortar la ejecución de las mismas y seguir ejecutando a partir de la instrucción siguiente a ellas. Se usa así:

```
break;
```

- *Instrucción continue*: Ya se ha visto que la instrucción **continue** sólo puede usarse dentro del bloque de instrucciones de una instrucción iterativa e indica que se desea pasar a reevaluar directamente la condición de la misma sin ejecutar el resto de instrucciones que contuviese. La evaluación de la condición se haría de la forma habitual: si es cierta se repite el bucle y si es falsa se continúa ejecutando por la instrucción que le sigue. Su sintaxis de uso es así de sencilla:

```
continue;
```

- *Instrucción return*: Esta instrucción se usa para indicar cuál es el objeto que ha de devolver un método. Su sintaxis es la siguiente:

```
return <objetoRetorno>;
```

La ejecución de esta instrucción provoca que se aborte la ejecución del método dentro del que aparece y que se devuelva el <objetoRetorno> al método que lo llamó. Como es lógico, este objeto ha de ser del tipo de retorno del método en que aparece el **return** o de alguno compatible con él, por lo que esta instrucción sólo podrá incluirse en métodos cuyo tipo de retorno no sea **void**, o en los bloques **get** de las propiedades o indizadores. De hecho, es obligatorio que todo método con tipo de retorno termine por un **return**.

Los métodos que devuelvan **void** pueden tener un **return** con una sintaxis especial en la que no se indica ningún valor a devolver sino que simplemente se usa **return** para indicar que se desea terminar la ejecución del método:

```
return;
```

- *Instrucción goto*: La instrucción **goto** permite pasar a ejecutar el código a partir de una instrucción cuya etiqueta se indica en el **goto**. La sintaxis de uso de esta instrucción es:

```
goto <etiqueta>;
```

Como en la mayoría de los lenguajes, el uso de ésta instrucción no se

Ing. Fausto Iocchi

recomienda porque dificulta innecesariamente la legibilidad del código y suele ser fácil simularla usando instrucciones iterativas y selectivas con las condiciones apropiadas. Sin embargo, en C# se incluye porque puede ser eficiente usarla si se anidan muchas instrucciones y para reducir sus efectos negativos se le han impuesto unas restricciones:

- Sólo se pueden etiquetar instrucciones, y no directivas preprocesador, directivas **using** o definiciones de miembros, tipos o espacios de nombres.
- La etiqueta indicada no pueda pertenecer a un bloque de instrucciones anidado dentro del bloque desde el que se usa el **goto** ni que etiquete a instrucciones de otro método diferente a aquél en el cual se encuentra el **goto** que la referencia.

Para etiquetar una instrucción de modo que pueda ser destino de un salto con **goto** basta precederla del nombre con el que se la quiera etiquetar seguido de dos puntos (**:**).

Por ejemplo, el siguiente código demuestra cómo usar **goto** y definir una etiqueta. Este programa de ejemplo lo que hace es mostrar por pantalla todos los argumentos que se le pasen como parámetros, aunque si alguno de fuese salir entonces se dejaría de mostrar argumentos y se aborta la ejecución de la aplicación. Véase además que este ejemplo pone de manifiesto una de las utilidades de la instrucción nula, ya que si no se hubiese escrito tras la etiqueta fin el programa no compilaría en tanto que toda etiqueta ha de preceder a alguna instrucción (aunque sea la instrucción nula).

```
using System;

class HolaMundoGoto {
  public static void Main(string[] args) {
    for (int i=0; i<args.Length; i++) {
      if (args[i] != "salir")
        Console.WriteLine(args[i]);
      else
        goto fin;
    }
fin: ;
  }
}
```

Nótese que, al fin y al cabo, los usos de **goto** dentro de instrucciones **switch** que se vieron al estudiar dicha instrucción, no son más que variantes del uso general de **goto**, ya que **default:** no es más que una etiqueta y **case** <valor>**:** puede verse como una etiqueta un tanto especial cuyo nombre es **case** seguido de espacios en blanco y un valor. En ambos casos, la etiqueta indicada ha de pertenecer al mismo **switch** que el **goto** usado y no vale que éste no la contenga pero la contenga algún **switch** que contenga al **switch** del **goto**.

Ing. Fausto Locatti

Unidad VI

Objetivo:

Previo repaso de la unidad anterior, más la lectura e investigación de otros materiales bibliográficos, y la inducción prestada por el facilitador, el participante empleará su conocimiento para realizar programas de computación más avanzados, utilizando tipos de datos compuestos, en los que resolverá problemas de mayor índole.

Contenido:

6. Arreglos.
   6.1. Arreglos unidimensionales.
   6.2. Arreglos anidados.
   6.3. Arreglos multidimensionales.
   6.4. La clase System.Array.
   6.5. Cadenas de texto.

Actividades del participante:

- Realizar lectura previa sobre el tema, utilizando la presente guía y otros medios bibliográficos.
- Repasar el tema sobre matrices (matemática).
- Interactuar en clase.
- Discutir en grupo.
- Realizar las actividades propuestas en clase.

Evaluación multidireccional:

- Evaluación formativa: a través de las discusiones de los conocimientos e intervenciones en clase.
- Evaluación sumativa: a través de un examen práctico y/o parcial.

Datos Compuestos: Arreglos

6. Arreglos

6.1. Arreglos unidimensionales

Un **arreglo unidimensional** o **vector** es un tipo especial de variable que es capaz de almacenar en su interior y de manera ordenada uno o varios datos de un determinado tipo. Para declarar arreglos se usa la siguiente sintaxis:

```
<tipoDatos>[] <nombreArreglo>;
```

Por ejemplo, un arreglo que pueda almacenar objetos de tipo **int** se declara así:

```
int[] arreglo;
```

Con esto el arreglo creado no almacenaría ningún objeto, sino que valdría **`null`**. Si se desea que verdaderamente almacene objetos hay que indicar cuál es el número de objetos que podrá almacenar, lo que puede hacerse usando la siguiente sintaxis al declararla:

```
<tipoDatos>[] <nombreArreglo> = new <tipoDatos>[<númeroDatos>];
```

Por ejemplo, un arreglo que pueda almacenar 100 objetos de tipo **int** se declara así:

```
int[] arreglo = new int[100];
```

Aunque también sería posible definir el tamaño del arreglo de forma separada a su declaración de este modo:

```
int[] arreglo;
arreglo = new int[100];
```

Ing. Fausto Iocchi

Con esta última sintaxis es posible cambiar dinámicamente el número de elementos de una variable de tipo arreglo sin más que irle asignando nuevos arreglos. Ello no significa que un arreglo se pueda redimensionar conservando los elementos que tuviese antes del cambio de tamaño, sino que ocurre todo lo contrario: cuando a una variable de tipo arreglo se le asigna un arreglo de otro tamaño, sus elementos antiguos son **sobrescritos** por los nuevos.

Si se crea una arreglo con la sintaxis hasta ahora explicada todos sus elementos tendrían el valor por defecto de su tipo de dato. Si queremos darles otros valores al declarar el arreglo, hemos de indicarlos entre llaves usando esta sintaxis:

<tipoDatos>**[]** <nombreArreglo> = **new** <tipoDatos>**[] {**<valores>**};**

Han de especificarse tantos <valores> como número de elementos se desee que tenga el arreglo, y si son más de uno se han de separar entre sí mediante comas (**,**) Nótese que ahora no es necesario indicar el número de elementos de el arreglo (aunque puede hacerse si se desea), pues el compilador puede deducirlo del número de valores especificados. Por ejemplo, para declarar un arreglo de cuatro elementos de tipo **int** con valores 5, 1, 4, 0 se podría hacer lo siguiente:

```
int[] arreglo = new int[] {5, 1, 4, 0};
```

Incluso se puede compactar aún más la sintaxis declarando el arreglo así:

```
int[] arreglo = {5, 1, 4, 0};
```

A la hora de acceder a los elementos almacenados en un arreglo basta indicar entre corchetes, y a continuación de la referencia a la misma, la posición que ocupe en el arreglo el elemento al que acceder. Cuando se haga hay que tener en cuenta que en C# las arreglos se indexan desde 0, lo que significa que el primer elemento de el arreglo ocupará su posición 0, el segundo ocupará la posición 1, y así sucesivamente para el resto de elementos. Por ejemplo, aunque es más ineficiente, el arreglo declarado en el último fragmento de código de ejemplo también podría haberse definido así:

```
int[] arreglo = new int[4];

arreglo[0] = 5;
arreglo[1]++;  // Por defecto se inicializó a 0, luego ahora
               // el valor de arreglo[1] pasa a ser 1
arreglo[2] = arreglo[0] – arreglo[1];  // arreglo[2] pasa a valer 4,
                                       // pues 5 - 4 = 1

// El contenido del arreglo será {5, 1, 4, 0}, pues arreglo[3]
// se inicializó por defecto a 0.
```

Hay que tener cuidado a la hora de acceder a los elementos de una arreglo ya que si se especifica una posición superior al número de elementos que pueda almacenar el arreglo se producirá una excepción o error de tipo **System.OutOfBoundsException**. Para evitar este tipo de excepciones (errores) puede consultar el valor del campo de sólo lectura **Length** que está asociado a todo arreglo y contiene el número de elementos de la misma. Por ejemplo, para asignar un 7 al último elemento del arreglo anterior se haría:

```
arreglo[arreglo.Length – 1] = 7;

// Se resta 1 porque arreglo.Length devuelve 4 pero el último
// elemento de el arreglo es arreglo[3]
```

6.2. Arreglos anidados

Un **arreglo anidado** no es más que un arreglo cuyos elementos son a su vez arreglos, pudiéndose así anidar cualquier número de arreglos. Para declarar arreglos de este tipo se usa una sintaxis muy similar a la explicada para los arreglos unidimensionales, sólo que ahora se indican tantos corchetes como nivel de anidación se desee.

Por ejemplo, para crear un arreglo de arreglos de elementos de tipo **int** formada por dos elementos, uno de los cuales fuese un arreglo de elementos de tipo **int** formada por los elementos de valores 1, 2 y el otro fuese un arreglo de elementos de tipo **int** y valores 3, 4, 5, se puede hacer:

Ing. Faustolocchi

```
int[][] dentado = new int[2][]{new int[] {1, 2}, new int[] {3, 4, 5}};
```

Como se indica explícitamente cuáles son los elementos del arreglo declarado no hace falta indicar el tamaño del mismo, por lo que la declaración anterior es equivalente a:

```
int[][] dentado = new int[][] {new int[] {1, 2}, new int[] {3, 4, 5}};
```

Es más, igual que como se vio con los arreglos unidimensionales también es válido hacer:

```
int[][] dentado = {new int[] {1, 2}, new int[] {3, 4, 5}};
```

Si no quisiésemos indicar cuáles son los elementos de los arreglos componentes, entonces tendríamos que indicar al menos cuál es el número de elementos que podrán almacenar (se inicializarán con valores por defecto) quedando:

```
int[][] dentado = {new int[2], new int[3]};
```

Si no queremos crear los arreglos componentes en el momento de crear el arreglo dentado, entonces tendremos que indicar por lo menos cuál es el número de arreglos componentes posibles (cada una valdría **null**), con lo que quedaría:

```
int[][] dentado = new int[2][];
```

Es importante señalar que no es posible especificar todas las dimensiones de un arreglo dentado en su definición si no se indica explícitamente el valor inicial de éstas entre llaves. **Es decir, esta declaración es incorrecta**:

```
int[][] dentado = new int[2][5];
```

Esto se debe a que el tamaño de cada arreglo componente puede ser distinto y con la sintaxis anterior no se puede decir cuál es el tamaño de cada una. Una opción hubiese sido considerar que es 5 para todas como se hace en Java, pero ello no se ha implementado en C# y habría que declarar el arreglo de, por ejemplo, esta manera:

```
int[][] dentado = {new int[5], new int[5]);
```

Finalmente, si sólo queremos declarar una variable de tipo arreglo dentado pero no queremos indicar su número de elementos, (luego la variable valdría **null**), entonces basta poner:

```
int[][] dentado;
```

Hay que precisar que aunque en los ejemplos hasta ahora presentes se han escrito ejemplos basados en arreglos dentados de sólo dos niveles de anidación, también es posible crear arreglos dentados de cualquier número de niveles de anidación. Por ejemplo, para una arreglo de arreglos de arreglos de enteros de 2 elementos en la que el primero fuese una arreglo dentado formada por dos arreglos de 5 enteros y el segundo elemento fuese una arreglo dentado formada por una arreglo de 4 enteros y otra de 3 se podría definir así:

```
int[][][] dentado = new int[][][] {new int[][] {new int[5],
                    new int[5]},new int[][] {new int[4], new int[3]}};
```

A la hora de acceder a los elementos de una arreglo dentado lo único que hay que hacer es indicar entre corchetes cuál es el elemento exacto de las arreglos componentes al que se desea acceder, indicándose un elemento de cada nivel de anidación entre unos corchetes diferentes pero colocándose todas las parejas de corchetes juntas y ordenadas de el arreglo más externa a la más interna. Por ejemplo, para asignar el valor 10 al elemento cuarto del arreglo que es elemento primero del arreglo que es elemento segundo del arreglo dentado declarada en último lugar se haría:

```
dentado[1][0][3] = 10;
```

6.3. Arreglos multidimensionales

Un **arreglo multidimensional** o **matriz** es aquella cuyos elementos se

Ing. Fausto Iocchi

encuentran organizados en una estructura de varias dimensiones. Para definirlas se utiliza una sintaxis similar a la usada para declarar arreglos unidimensionales pero separando las diferentes dimensiones mediante comas (**,**) Por ejemplo, una arreglo multidimensional de elementos de tipo **int** que conste de 12 elementos puede tener sus elementos distribuidos en dos dimensiones formando una estructura 3x4 similar a una matriz de la forma:

```
1   2   3   4
5   6   7   8
9  10  11  12
```

Este arreglo se podría declarar así:

```
int[,] multidim = new int[3,4] {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
```

En realidad no es necesario indicar el número de elementos de cada dimensión del arreglo ya que pueden deducirse de los valores explícitamente indicados entre llaves, por lo que la definición anterior es similar a esta:

```
int[,] multidim = new int[,] {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
```

Incluso puede reducirse aún más la sintaxis necesaria quedando tan sólo:

```
int[,] multidim = {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
```

Si no queremos indicar explícitamente los elementos del arreglo al declararla, podemos obviarlos pero aún así indicar el tamaño de cada dimensión del arreglo (a los elementos se les daría el valor por defecto de su tipo de dato) así:

```
int[,] arregloMultidimensional = new int[3, 4];
```

También podemos no especificar ni siquiera el número de elementos del arreglo de esta forma (arregloMultidimensional contendría ahora **null**):

```
int[,] arregloMultidimensional;
```

Nótese que tanto las arreglos dentados como las arreglos multidimensionales pueden ser utilizadas tanto para representar estructuras matriciales como para, en general, representar cualquier estructura de varias dimensiones.

Como los arreglos dentados son arreglos de arreglos, cada uno de sus elementos puede ser un arreglo de un tamaño diferente. Así, con las arreglos dentados podemos representar matrices en las que cada columna tenga un tamaño distinto (por el aspecto "aserrado" de este tipo de matrices es por lo que se les llama arreglos dentados), mientras que usando arreglos multidimensionales sólo es posible crear matrices rectangulares o cuadradas.

Las estructuras aserradas pueden simularse usando matrices multidimensionales con todas sus columnas del tamaño de la columna más grande necesaria, aunque ello implica desperdiciar mucha memoria sobre todo si los tamaños de cada columna son muy diferentes y el arreglo es grande. De todos modos, las estructuras más comunes que se usan en la mayoría de aplicaciones suelen ser rectangulares o cuadradas.

Los tiempos que se tardan en crear y destruir arreglos dentados son superiores a los que se tardan en crear y destruir arreglos multidimensionales. Esto se debe a que las primeras son arreglos de arreglos mientras que las segundas son un único arreglo.

Por ejemplo, para crear una arreglo dentado [100][100] hay que crear 101 arreglos (el arreglo dentado más las 100 arreglos que contiene), mientras que para crear una crear una arreglo bidimensional [100, 100] hay que crear una única arreglo.

6.4. La clase System.Array

En realidad, todos los arreglos que definamos, sea cual sea el tipo de elementos que contengan, son objetos que derivan de **System.Array**. Es

Ing. Fausto Inocchi

decir, van a disponer de todos los miembros que se han definido para esta clase, entre los que son destacables:

6.4.1. **Length**: Campo de sólo lectura que informa del número total de elementos que contiene el arreglo. Si el arreglo tiene más de una dimensión o nivel de anidación indica el número de elementos de todas sus dimensiones y niveles. Por ejemplo:

```
int[] arreglo = {1, 2, 3, 4};
int[][] arreglo2 = {new int[] {1, 2}, new int[] {3, 4, 5}};
int[,] arreglo3 = {{1, 2}, {3, 4, 5, 6}};

Console.WriteLine(arreglo.Length);   // Imprime 4
Console.WriteLine(arreglo2.Length);  // Imprime 5
Console.WriteLine(arreglo3.Length);  // Imprime 6
```

6.4.2. **int GetLength(int dimensión)**: Método que devuelve el número de elementos de la dimensión especificada. Las dimensiones se indican empezando a contar desde cero, por lo que si quiere obtenerse el número de elementos de la primera dimensión habrá que usar GetLength(0), si se quiere obtener los de la segunda habrá que usar GetLength(1), etc. Por ejemplo:

```
int[,] arreglo = {{1, 2}, {3, 4, 5, 6}};

Console.WriteLine(arreglo.GetLength(0));    // Imprime 2
Console.WriteLine(<garreglo.GetLength(1));  // Imprime 4
```

6.4.3. **void CopyTo(Array destino, int posición)**: Copia todos los elementos del arreglo sobre la que se aplica en el arreglo **destino** a partir de la **posición** que está indicada. Por ejemplo:

```
int[] arreglo1 = {1, 2, 3, 4};
int[] arreglo2 = {5, 6, 7, 8, 9};

arreglo1.CopyTo(arreglo2, 1);
// A partir de ahora, arreglo2 contendrá {5, 1, 2, 3, 4}
```

Ambos arreglos **deben ser unidimensionales**, el arreglo de destino ha de ser de un tipo que pueda almacenar los objetos de el arreglo origen, el índice especificado ha de ser válido (mayor o igual que cero y menor que el tamaño de el arreglo de destino) y no ha de valer **null** ninguno de las arreglos.

6.5. Cadenas de texto

Una **cadena de texto** no es más que una secuencia de caracteres. Microsoft .NET las representa internamente en formato Unicode, y C# las representan externamente como objetos de un tipo de dato **string**, que no es más que un alias del tipo **System.String**.

Algunos lenguajes de programación no poseen una definición para este tipo de datos en particular (al menos no de forma nativa), y los que sí lo tienen, no la tienen definida como una clase.

Las cadenas de texto suelen crearse a partir literales de cadena o de otras cadenas previamente creadas. Ejemplos de ambos casos se muestran a continuación:

```
string cadena1 = "José Antonio";
string cadena2 = cadena1;
```

En el primer caso se ha creado un objeto **string** que representa a la cadena formada por la secuencia de caracteres José Antonio indicada literalmente (nótese que las comillas dobles entre las que se encierran los literales de cadena no forman parte del contenido de la cadena que representan sino que sólo se usan como delimitadores de la misma) En el segundo caso la variable cadena2 creada se genera a partir de la variable cadena1 ya existente, por lo que ambas variables apuntarán al mismo objeto en memoria.

Hay que tener en cuenta que el tipo **string** es un tipo de variable de referencia, por lo que en principio la comparación entre objetos de este tipo debería comparar sus direcciones de memoria, como pasa con cualquier tipo referencia. Sin embargo, si ejecutamos el siguiente código veremos que esto no ocurre en el caso de las cadenas:

Ing. Fausto Iocchi

```
using System;

public class IgualdadCadenas {
  public static void Main() {
    string cadena1 = "José Antonio";
    string cadena2 = String.Copy(cadena1);

    Console.WriteLine(cadena1 == cadena2);
  }
}
```

El método **Copy()** de la clase **String** usado devuelve una copia del objeto que se le pasa como parámetro. Por tanto, al ser objetos diferentes se almacenarán en posiciones distintas de memoria y al compararlos debería devolverse **false** como pasa con cualquier tipo referencia. Sin embargo, si ejecuta el programa verá que lo que se obtiene es precisamente lo contrario: **true**. Esto se debe a que para hacer más intuitivo el trabajo con cadenas, en el lenguaje C# se ha modificado el operador de igualdad para que cuando se aplique entre cadenas se considere que sus operandos son iguales sólo si son lexicográficamente equivalentes y no se referencian al mismo objeto en memoria. Además, esta comparación se hace teniendo en cuenta la capitalización usada, por lo que (`"Hola" == "HOLA"`) ó (`"Hola" == "hola"`) devolverán **false** ya que contienen las mismas letras pero con distinta capitalización.

Al igual que el significado del operador **==** ha sido especialmente modificado para trabajar con cadenas, lo mismo ocurre con el operador binario **+**. En este caso, cuando se aplica entre dos cadenas o una cadena y un carácter lo que hace es devolver una nueva cadena con el resultado de concatenar sus operandos. Así por ejemplo, en el siguiente código las dos variables creadas almacenarán la cadena Hola Mundo:

```
public class Concatenación {
  public static void Main() {
    string cadena = "Hola" + " Mundo";
    string cadena2 = "Hola Mund" + 'o';
  }
}
```

Por otro lado, el acceso a las cadenas se hace de manera similar a como si de arreglos de caracteres se tratase: su campo **Length** almacenará el número de caracteres que la forman y para acceder a sus elementos se utiliza el operador **[]**. Por ejemplo, el siguiente código muestra por pantalla cada carácter de la cadena Hola en una línea diferente:

```
using System;

public class AccesoCadenas
{
  public static void Main()
  {
    string cadena = "Hola";

    Console.WriteLine(cadena[0]);
    Console.WriteLine(cadena[1]);
    Console.WriteLine(cadena[2]);
    Console.WriteLine(cadena[3]);
  }
}
```

Sin embargo, hay que señalar una diferencia importante respecto a la forma en que se accede a los arreglos: las cadenas son inmutables, lo que significa que no es posible modificar los caracteres que las forman. Esto se debe a que el compilador comparte en memoria las referencias a literales de cadena lexicográficamente equivalentes para así ahorrar memoria, y si se permitiese modificarlos los cambios que se hiciesen a través de una variable a una cadena compartida afectarían al resto de variables que la compartan, lo que podría causar errores difíciles de detectar. Por tanto, hacer esto es incorrecto:

```
string cadena = "Hola";
cadena[0] = "A"; // Error: No se pueden modificar las cadenas
```

Sin embargo, el hecho de que no se puedan modificar las cadenas no significa que no se puedan cambiar los objetos almacenados en las variables de tipo **string**. Por ejemplo, el siguiente código es válido:

```
string cad = "Hola";
cad = "Adios";  // Correcto, pues no se modifica la cadena almacenada
                // en cad sino que se hace que cad pase a almacenar
               // otra cadena distinta.
```

Aparte de los métodos ya vistos, en la clase **System.String** se definen

Ing. Fausto Iocchi

muchos otros métodos aplicables a cualquier cadena y que permiten manipularla. Los principales son:

6.5.1. **int IndexOf(string subcadena)**: Indica cuál es el índice de la primera aparición de la subcadena indicada dentro de la cadena sobre la que se aplica. La búsqueda de dicha subcadena se realiza desde el principio de la cadena, pero es posible indicar en un segundo parámetro opcional de tipo **int** cuál es el índice de la misma a partir del que se desea empezar a buscar. Del mismo modo, la búsqueda acaba al llegar al final de la cadena sobre la que se busca, pero pasando un tercer parámetro opcional de tipo **int** es posible indicar algún índice anterior donde terminarla.

Nótese que es un método muy útil para saber si una cadena contiene o no alguna subcadena determinada, pues sólo si no la encuentra devolverá un valor igual a **–1**.

6.5.2. **int LastIndexOf(string subcadena)**: Funciona de forma similar a **IndexOf()** sólo que devuelve la posición de la última aparición de la subcadena buscada en lugar de devolver la de la primera.

6.5.3. **string Insert(int posición, string subcadena)**: Devuelve la cadena resultante de insertar la subcadena indicada en la posición especificada de la cadena sobre la que se aplica.

6.5.4. **string Remove(int posición, int número)**: Devuelve la cadena resultante de eliminar el número de caracteres indicado que hubiese en la cadena sobre al que se aplica a partir de la posición especificada.

6.5.5. **string Replace(string aSustituir, string sustituta)**: Devuelve la cadena resultante de sustituir en la cadena sobre la que se aplica toda aparición de la cadena a Sustituir indicada por la cadena sustituta especificada como segundo parámetro.

6.5.6. **string Substring(int posición, int número)**: Devuelve la subcadena de la cadena sobre la que se aplica que comienza en la posición indicada y tiene el número de caracteres especificados. Si no se indica dicho número se devuelve la subcadena que va desde la posición indicada hasta el final de la cadena.

6.5.7. **string ToUpper()** y **string ToLower()**: Devuelven, respectivamente, la cadena que resulte de convertir a mayúsculas o minúsculas la cadena sobre la que se aplican.

Es preciso incidir en que aunque hayan métodos de inserción, reemplazo o eliminación de caracteres que puedan dar la sensación de que es posible modificar el contenido de una cadena, en realidad las cadenas son inmutables y dicho métodos lo que hacen es devolver una nueva cadena con el contenido correspondiente a haber efectuado las operaciones de modificación solicitadas sobre la cadena a la que se aplican. Por ello, las cadenas sobre las que se aplican quedan intactas como muestra el siguiente ejemplo:

```
using System;

public class EjemploInmutabilidad
{
  public static void Main()
  {
    string cadena1 = "Hola";
    string cadena2 = cadena1.Remove(0, 1);

    Console.WriteLine(cadena1);
    Console.WriteLine(cadena2);
  }
}
```

La salida por pantalla de este ejemplo demuestra lo antes dicho, pues es:

```
Hola
ola
```

Como se ve, tras el **Remove()** la cadena1 permanece intacta y el contenido de cadena2 es el que debería tener cadena1 si se le hubiese eliminado su primer carácter.

Ing. Fausto Cocchi

Unidad VII

Objetivo:

Al finalizar esta unidad, el participante aplicará el conocimiento obtenido para realizar cualquier programa de computación, siguiendo el paradigma de la programación orientada a objetos.

Contenido:

7. Clases y Objetos.
   - 7.1. Conceptos de clase y objeto.
   - 7.2. Creación de objetos (Operador new).
   - 7.3. Constructor por defecto.
   - 7.4. Referencia al objeto actual con this.
   - 7.5. Encapsulación.
   - 7.6. Métodos o Procedimientos.
   - 7.7. Tipos de parámetros. Sintaxis de definición.
   - 7.8. Constructores.
   - 7.9. Destructores.
   - 7.10. Propiedades.
   - 7.11. Herencia.
   - 7.12. Estructuras.

Actividades del participante:

- Realizar lectura previa sobre el tema, utilizando la presente guía y otros medios bibliográficos.
- Interactuar en clase.
- Discutir en grupo.
- Realizar las actividades propuestas en clase.

Evaluación multidireccional:

- Evaluación formativa: a través de las discusiones de los conocimientos e intervenciones en clase.
- Evaluación sumativa: a través de un examen práctico y/o parcial.

Clases y Objetos

7. Clases y Objetos

7.1. Conceptos de clase y objeto

C# es un lenguaje orientado a objetos puro, lo que significa que todo con lo que vamos a trabajar (y lo hemos estado haciendo), en este lenguaje, son objetos. Un **objeto** es un agregado de datos y de métodos que permiten manipular dichos datos, y un programa no es más que un conjunto de objetos que interaccionan unos con otros a través de sus métodos.

Una **clase** es la definición de las características concretas de un determinado tipo de objetos. Es decir, de cuáles son los datos y los métodos de los que van a disponer todos los objetos de ese tipo. Por esta razón, se suele decir que el **tipo de dato** de un objeto es la clase que define las características del mismo.

7.1.1. Sintaxis de definición de clases

La sintaxis básica para definir una clase es la que a continuación se muestra:

```
class <nombreClase>
{
  <miembros>
}
```

De este modo se definiría una clase de nombre <nombreClase> cuyos miembros son los definidos en <miembros>. Los **miembros** de una clase son los datos y métodos de los que van a disponer todos los objetos de la misma. Un ejemplo de cómo declarar una clase de nombre A que no tenga ningún miembro es la siguiente:

```
class A { }
```

Ing. Fausto Iocchi

Una clase así declarada no dispondrá de ningún miembro a excepción de los implícitamente definidos de manera común para todos los objetos que creemos en C#.

Aunque en C# hay muchos tipos de miembros distintos, por ahora vamos a considerar que éstos únicamente pueden ser campos (atributos) o métodos, y vamos a hablar un poco acerca de ellos y de cómo se definen.

- **Campos (Atributos):** Un **campo** es un dato común a todos los objetos de una determinada clase. Para definir cuáles son los campos de los que una clase dispone se usa la siguiente sintaxis dentro de la zona señalada como <miembros> en la definición de la misma:

```
<tipoCampo> <nombreCampo>;
```

El nombre que demos al campo puede ser cualquier identificador que queramos siempre y cuando siga las reglas descritas en el *Tema: Aspectos Léxicos* para la escritura de identificadores y no coincida con el nombre de ningún otro miembro previamente definido en la definición de clase.

Los campos de un objeto son a su vez objetos, y en <tipoCampo> hemos de indicar cuál es el tipo de dato del objeto que vamos a crear. Éste tipo puede corresponderse con cualquiera que los predefinidos en la BCL o con cualquier otro que nosotros hayamos definido siguiendo la sintaxis arriba mostrada. A continuación se muestra un ejemplo de definición de una clase de nombre Persona que dispone de tres campos:

```
class Persona
{
  string Nombre; // Campo que almacena su nombre
  int Edad;      // Campo que almacena su edad
  int Cedula;    // Campo que almacena su cédula
}
```

Según esta definición, todos los objetos de clase Persona incorporarán campos que almacenarán cuál es el nombre de la persona que cada objeto representa, cuál es su edad y cuál es su cédula. El tipo **int** incluido en la definición del campo Edad es un tipo predefinido en la BCL cuyos objetos son capaces de almacenar números enteros con signo comprendidos entre -2.147.483.648 y 2.147.483.647 (32 bits), mientras que **string** es un tipo predefinido que permite almacenar cadenas de texto que sigan el formato de los literales de cadena visto en el Tema *Aspectos Léxicos*. Para acceder a un campo de un determinado objeto se usa la sintaxis:

```
<objeto>.<campo>
```

Por ejemplo, para acceder al campo Edad de un objeto Persona llamado p y cambiar su valor por 20 se haría:

```
p.Edad = 20;
```

- **Métodos:** Un **método** es un conjunto de instrucciones a las que se les asocia un nombre de modo que si se desea ejecutarlas basta referenciarlas a través de dicho nombre en vez de tener que escribirlas. Dentro de estas instrucciones es posible acceder con total libertad a la información almacenada en los campos pertenecientes a la clase dentro de la que el método se ha definido, por lo que como al principio del tema se indicó, los métodos permiten manipular los datos almacenados en los objetos.

La sintaxis que se usa en C# para definir los métodos es la siguiente:

```
<tipoDevuelto> <nombreMétodo>(<parametros>)
{
  <instrucciones>
}
```

Todo método puede devolver un objeto como resultado de la ejecución de las instrucciones que lo forman, y el tipo de dato al que pertenece este objeto es lo que se indica en <tipoDevuelto>. Si no devuelve nada se indica **void**, y si devuelve algo es obligatorio finalizar la ejecución de sus instrucciones con alguna instrucción **return** <objeto>**;** que indique qué objeto ha de devolverse.

Ing. Fausto Jocelli

Opcionalmente todo método puede recibir en cada llamada una lista de objetos a los que podrá acceder durante la ejecución de sus instrucciones. En *<parametros>* se indica es cuáles son los tipos de dato de estos objetos y cuál es el nombre con el que harán referencia las instrucciones del método a cada uno de ellos. Aunque los objetos que puede recibir el método pueden ser diferentes cada vez que se solicite su ejecución, siempre han de ser de los mismos tipos y han de seguir el orden establecido en *<parametros>*.

Un ejemplo de cómo declarar un método de nombre Cumpleaños es la siguiente modificación de la definición de la clase Persona usada antes como ejemplo:

```
class Persona
{
  string Nombre;   // Campo que almacena su nombre
  int Edad;        // Campo que almacena su edad
  string Cedula;   // Campo que almacena su cédula

  // Incrementa en uno de la edad del objeto Persona
  void Cumpleaños()
  {
    Edad++;
  }
}
```

La sintaxis usada para llamar a los métodos de un objeto es la misma que la usada para llamar a sus campos, sólo que ahora tras el nombre del método al que se desea llamar hay que indicar entre paréntesis cuáles son los valores que se desea dar a los parámetros del método al hacer la llamada. O sea, se escribe:

```
<objeto>.<método>(<parámetros>)
```

Como es lógico, si el método no tomase parámetros se dejarían vacíos los parámetros en la llamada al mismo. Por ejemplo, para llamar al método Cumpleaños() de un objeto Persona llamado p se haría:

```
// El método no toma parámetros, luego no le pasamos ninguno
p.Cumpleaños();
```

Es importante señalar que en una misma clase pueden definirse varios métodos con el **mismo nombre** siempre y cuando tomen diferente número o tipo de parámetros. A esto se le conoce como **sobrecarga de métodos**, y es posible ya que el compilador sabrá a cual llamar a partir de los *<parámetros>* especificados.

Sin embargo, lo que no se permite es definir varios métodos que únicamente se diferencien en su valor de retorno, ya que como éste no se tiene porqué indicar al llamarlos no podría diferenciarse a que método en concreto se hace referencia en cada llamada. Por ejemplo, a partir de la llamada:

```
p.Cumpleaños();
```

Si además de la versión de Cumpleaños() que no retorna nada hubiese otra que retornase un **int**, ¿cómo sabría entonces el compilador a cuál llamar?

Antes de continuar es preciso señalar que en C# todo, incluido los literales, son objetos del tipo de cada literal y por tanto pueden contar con miembros a los que se accedería tal y como se ha explicado. Para entender esto no hay nada mejor que un ejemplo:

```
string s = 12.ToString();
```

Este código almacena el literal de cadena "12" en la variable s, pues 12 es un objeto de tipo **int** (tipo que representa enteros) y cuenta con el método común a todos los **int**s llamado **ToString()** que lo que hace es devolver una cadena cuyos caracteres son los dígitos que forman el entero representado por el **int** sobre el que se aplica; y como la variable s es de tipo **string** (tipo que representa cadenas) es perfectamente posible almacenar dicha cadena en ella, que es lo que se hace en el código anterior.

7.2. Creación de objetos (Operador new)

Ahora que ya sabemos cómo definir las clases de objetos que podremos usar en nuestras aplicaciones ha llegado el momento de explicar cómo crear objetos de una determinada clase. Algo de ello ya se introdujo en el

Ing. Fausto Iocchi

tema *Aspectos Léxicos* cuando se comentó la utilidad del operador **new**, que precisamente es crear objetos y cuya sintaxis es:

```
new <nombreTipo>(<parametros>)
```

Este operador crea un nuevo objeto del tipo cuyo nombre se le indica y llama durante su proceso de creación al constructor del mismo apropiado según los valores que se le pasen en <parametros>, devolviendo una referencia al objeto recién creado. Hay que resaltar el hecho de que **new** no devuelve el propio objeto creado, sino una referencia a la dirección de memoria dinámica donde en realidad se ha creado.

El antes comentado **constructor** de un objeto no es más que un método definido en la definición de su tipo que tiene el mismo nombre que la clase a la que pertenece el objeto y no tiene valor de retorno. Como **new** siempre devuelve una referencia a la dirección de memoria donde se cree el objeto y los constructores sólo pueden usarse como operandos de **new**, no tiene sentido que un constructor devuelva objetos, por lo que no tiene sentido incluir en su definición un campo <tipoDevuelto> y el compilador considera erróneo hacerlo (aunque se indique **void**).

El constructor recibe ese nombre debido a que su código suele usarse precisamente para construir el objeto, para inicializar sus miembros. Por ejemplo, a nuestra clase de ejemplo Persona le podríamos añadir un constructor dejándola así:

```
class Persona {
  string Nombre;    // Campo que almacena su nombre
  int Edad;         // Campo que almacena su edad
  int Cedula;       // Campo que almacena su cedula

  void Cumpleaños() {  // Incrementa en uno la edad del objeto Persona
    Edad++;
  }

  Persona(string nombre, int edad, int cedula) { // Constructor
    Nombre = nombre;
    Edad = edad;
    Cedula = cedula;
  }
}
```

Como se ve en el código, el constructor toma como parámetros los valores con los que deseemos inicializar el objeto a crear. Gracias a él, podemos crear un objeto Persona de nombre José, de 22 años de edad y cedula 12344321 así:

```
new Persona("José", 22, 12344321)
```

Nótese que la forma en que se pasan parámetros al constructor consiste en indicar los valores que se ha de dar a cada uno de los parámetros indicados en la definición del mismo separándolos por comas. Obviamente, si un parámetro se definió como de tipo **string** habrá que pasarle una cadena, si se definió de tipo **int** habrá que pasarle un entero y, en general, a todo parámetro habrá que pasarle un valor de su mismo tipo (o de alguno convertible al mismo), produciéndose un error al compilar si no se hace así.

En realidad un objeto puede tener múltiples constructores, aunque para diferenciar a unos de otros es obligatorio que se diferencien en el número u orden de los parámetros que aceptan, ya que el nombre de todos ellos ha de coincidir con el nombre de la clase de la que son miembros. De ese modo, cuando creemos el objeto el compilador podrá inteligentemente determinar cuál de los constructores ha de ejecutarse en función de los valores que le pasemos al **new**.

Una vez creado un objeto lo más normal es almacenar la dirección devuelta por **new** en una variable del tipo apropiado para el objeto creado. El siguiente ejemplo -que como es lógico irá dentro de la definición de algún método- muestra cómo crear una variable de tipo Persona llamada p y cómo almacenar en ella la dirección del objeto que devolvería la anterior aplicación del operador **new**:

```
// Creamos variable p
Persona p;

// Almacenamos en p el objeto creado con new
p = new Persona("Jose", 22, 12344321);
```

A partir de este momento la variable p contendrá una referencia a un

Ing. Fausto Tuschi

objeto de clase Persona que representará a una persona llamada José de 22 años y cédula 12344321-A. O lo que prácticamente es lo mismo y suele ser la forma comúnmente usada para decirlo: la variable p representa a una persona llamada José de 22 años y cédula 12344321-A.

Como lo más normal suele ser crear variables donde almacenar referencias a objetos que creemos, las instrucciones anteriores pueden compactarse en una sola así:

```
Persona p = new Persona("José", 22, 12344321);
```

De hecho, una sintaxis más general para la definición de variables es la siguiente:

```
<tipoDato> <nombreVariable> = <valorInicial>;
```

La parte **= <valorInicial>** de esta sintaxis es en realidad opcional, y si no se incluye la variable declarada pasará a almacenar una referencia nula (contendrá el literal **null**).

7.3. Constructor por defecto

No es obligatorio definir un constructor para cada clase, y en caso de que no definamos ninguno el compilador creará uno por nosotros sin parámetros ni instrucciones. Es decir, como si se hubiese definido de esa forma:

```
<nombreTipo>() {
}
```

Gracias a este constructor introducido automáticamente por el compilador, si Coche es una clase en cuya definición no se ha incluido ningún constructor, siempre será posible crear uno nuevo usando el operador **new** así:

```
// Crea coche c llamando al constructor por defecto de Coche
Coche c = new Coche();
```

Hay que tener en cuenta una cosa: el constructor por defecto es sólo incluido por el compilador si no hemos definido ningún otro constructor. Por tanto, si tenemos una clase en la que hayamos definido algún constructor con parámetros pero ninguno sin parámetros no será válido crear objetos de la misma llamando al constructor sin parámetros, pues el compilador no lo habrá definido automáticamente. Por ejemplo, con la última versión de la clase de ejemplo Persona es inválido hacer:

```
// ERROR: El único constructor de Persona toma 3 parámetros
Persona p = new Persona();
```

7.4. Referencia al objeto actual con this

Dentro del código de cualquier método de un objeto siempre es posible hacer referencia al propio objeto usando la palabra reservada **this**. Esto puede venir bien a la hora de escribir constructores de objetos debido a que permite que los nombres que demos a los parámetros del constructor puedan coincidir nombres de los campos del objeto sin que haya ningún problema. Por ejemplo, el constructor de la clase Persona escrito anteriormente se puede rescribir así usando **this**:

```
Persona(string Nombre, int Edad, int Cedula) {
  this.Nombre = Nombre;
  this.Edad = Edad;
  this.Cedula = Cedula;
}
```

Es decir, dentro de un método con parámetros cuyos nombres coincidan con campos, se da preferencia a los parámetros y para hacer referencia a los campos hay que prefijarlos con el **this** tal y como se muestra en el ejemplo.

El ejemplo anterior puede que no resulte muy interesante debido a que para evitar tener que usar **this** podría haberse escrito el constructor tal y como se mostró en la primera versión del mismo: dando nombres que empiecen en minúscula a los parámetros y nombres que empiecen con mayúsculas a los campos. De hecho, ese es el convenio que Microsoft recomienda usar. Sin embargo, como más adelante se verá sí que puede ser útil **this** cuando los campos a inicializar sean privados, ya que el

Ing. Fausto Locatti

convenio de escritura de identificadores para campos privados recomendado por Microsoft coincide con el usado para dar identificadores a parámetros (obviamente otra solución sería dar cualquier otro nombre a los parámetros del constructor o los campos afectados, aunque así el código perdería algo legibilidad).

Un uso más frecuente de **this** en C# es el de permitir realizar llamadas a un método de un objeto desde código ubicado en métodos del mismo objeto. Es decir, en C# siempre es necesario que cuando llamemos a algún método de un objeto precedamos al operador "**.**" de alguna expresión que indique cuál es el objeto a cuyo método se desea llamar, y si éste método pertenece al mismo objeto que hace la llamada la única forma de conseguir indicarlo en C# es usando **this**.

7.5. Encapsulación

La **encapsulación** es un mecanismo que permite a los diseñadores de tipos de datos determinar qué miembros de los tipos creen pueden ser utilizados por otros programadores y cuáles no.

La encapsulación se consigue añadiendo **modificadores de acceso** en las definiciones de miembros y tipos de datos. Estos modificadores son partículas que se les colocan delante para indicar desde qué códigos puede accederse a ellos, entendiéndose por acceder el hecho de usar su nombre para cualquier cosa que no sea definirlo, como llamarlo si es una función, leer o escribir su valor si es un campo, crear objetos o heredar de él si es una clase, etc.

Por defecto se considera que los miembros de un tipo de dato sólo son accesibles desde código situado dentro de la definición del mismo, aunque esto puede cambiarse precediéndolos de uno los siguientes modificadores (aunque algunos de ellos ya se han explicado a lo largo del tema, aquí se recogen todos de manera detallada) al definirlos:

- **public:** Puede ser accedido desde cualquier código.
- **protected:** Desde una clase sólo puede accederse a miembros protected de objetos de esa misma clase o de subclases suyas.

Obviamente siempre que se herede de una clase se tendrá total acceso en la clase hija e implícitamente sin necesidad de usar la sintaxis <objeto>**.**<miembro>- a los miembros que ésta herede de su clase padre, como muestra el siguiente ejemplo:

```
using System;

class A {
  protected int x = 5;
}

class B: A {
  B(){
    Console.WriteLine("Heredado x={0} de clase A", x);
  }

  public static void Main() {
    new B();
  }
}
```

Como es de esperar, la salida por pantalla del programa de ejemplo será:

```
Heredado x=5 de clase A
```

A lo que no se podrá acceder desde una clase hija es a los miembros protegidos de otros objetos de su clase padre, sino sólo a los heredados. Es decir:

```
using System;

class A {
  protected int x = 5;
}

class B: A {
  B(A objeto) {
    Console.WriteLine("Heredado x={0} de clase A", x);
    Console.WriteLine(objeto.x); // Error, no es el x heredado
  }

  public static void Main() {
    new B(new A());
  }
}
```

Ing. Fausto Iocchi

- **private:** Sólo puede ser accedido desde el código de la clase a la que pertenece. Es lo considerado por defecto.

Si se duda sobre el modificador de visibilidad a poner a un miembro, es mejor ponerle inicialmente el que proporcione menos permisos de accesos, ya que si luego detecta que necesita darle más permisos siempre podrá cambiárselo por otro menos restringido. Sin embargo, si se le da uno más permisivo de lo necesario y luego se necesita cambiar por otro menos permisivo, los códigos que escrito en base a la versión más permisiva que dependiesen de dicho miembro podrían dejar de funcionar por quedarse sin acceso a él.

7.6. Métodos o Procedimientos

7.6.1. Concepto de Método

Un **método** es un conjunto de instrucciones a las que se les da un determinado nombre de tal manera que sea posible ejecutarlas en cualquier momento sin tenerlas que rescribir sino usando sólo su nombre. A estas instrucciones se les denomina **cuerpo** del método, y a su ejecución a través de su nombre se le denomina **llamada** al método.

La ejecución de las instrucciones de un método puede producir como resultado un objeto de cualquier tipo. A este objeto se le llama **valor de retorno** del método y es completamente opcional, pudiéndose escribir métodos que no devuelvan ninguno.

La ejecución de las instrucciones de un método puede depender del valor de unas variables especiales denominadas **parámetros** del método, de manera que en función del valor que se dé a estas variables en cada llamada la ejecución del método se pueda realizar de una u otra forma y podrá producir uno u otro valor de retorno.

7.6.2. Definición de un Método

Para definir un método hay que indicar tanto cuáles son las instrucciones que forman su cuerpo como cuál es el nombre que se le dará, cuál es el tipo de objeto que puede devolver y cuáles son los parámetros que puede tomar. Esto se indica definiéndolo así:

```
<tipoRetorno> <nombreMétodo>(<parámetros>)
{
  <cuerpo>
}
```

En <tipoRetorno> se indica cuál es el tipo de dato del objeto que el método devuelve, y si no devuelve ninguno se ha de escribir **void** en su lugar.

Como nombre del método se puede poner en <nombreMétodo> cualquier identificador válido.

Aunque es posible escribir métodos que no tomen parámetros, si un método los toma se ha de indicar en *<parámetros>* cuál es el nombre y tipo de cada uno, separándolos con comas si son más de uno y siguiendo la sintaxis que más adelante se explica.

El *<cuerpo>* del método también es opcional, pero si el método retorna algún tipo de objeto entonces ha de incluir al menos una instrucción **return** que indique cuál objeto.

A continuación se muestra un ejemplo de cómo definir un método de nombre Saluda cuyo cuerpo consista en escribir en la consola el mensaje "Hola Mundo" y que devuelva un objeto **int** de valor 1:

```
int Saluda()
{
  Console.WriteLine("Hola Mundo");
  return 1;
}
```

7.6.3. Llamadas a Métodos

La forma en que se puede llamar a un método depende del tipo de método del que se trate. Si es un **método de objeto** (método no estático) se ha de usar la notación:

Ing. Fausto Iocchi

```
<objeto>.<nombreMétodo>(<valoresParámetros>)
```

El <objeto> indicado puede ser directamente una variable del tipo de datos al que pertenezca el método o puede ser una expresión que produzca como resultado una variable de ese tipo.

En caso de que sea un **método de tipo** (método estático), entones se ha de usar (como ejemplo, se puede tomar la clase *Console* de C#, y los métodos comúnmente utilizados: *ReadLine* y *WriteLine*):

```
<tipo>.<nombreMétodo>(<valoresParámetros>)
```

Ahora en <tipo> ha de indicarse el tipo donde se haya definido el método o algún subtipo suyo. Sin embargo, si el método pertenece al mismo tipo que el código que lo llama entonces se puede usar la notación abreviada:

```
<nombreMétodo>(<valoresParámetros>)
```

El formato en que se pasen los valores a cada parámetro en <valoresParámetros> a aquellos métodos que tomen parámetros depende del tipo de parámetro que sea. Esto se explica en el siguiente apartado.

7.7. Tipos de parámetros. Sintaxis de definición

La forma en que se define cada parámetro de un método depende del tipo de parámetro del que se trate. En C# se admiten cuatro tipos de parámetros: parámetros de entrada, parámetros de salida y parámetros por referencia.

7.7.1. Parámetros de entrada

Un **parámetro de entrada** recibe una copia del valor que almacenaría una variable del tipo del objeto que se le pase. Por tanto, si el objeto es de un tipo valor se le pasará una copia del objeto y cualquier modificación que se haga al parámetro dentro del cuerpo del método no afectará al objeto original sino a su copia; mientras que si el objeto es de un tipo referencia entonces se le pasará una copia de la referencia al mismo y cualquier modificación que se haga al parámetro dentro del método también afectará al objeto original ya que en realidad el parámetro referencia a ese mismo objeto original.

Para definir un parámetro de entrada basta indicar cuál el nombre que se le desea dar y el cuál es tipo de dato que podrá almacenar. Para ello se sigue la siguiente sintaxis:

```
<tipoParámetro> <nombreParámetro>
```

Por ejemplo, el siguiente código define un método llamado **`Suma`** que toma dos parámetros de entrada de tipo **`int`** llamados **par1** y **par2**, y devuelve un **`int`** el cual será la suma entre estos dos valores representados por ambos parámetros:

```
int Suma(int par1, int par2) {
  return (par1 + par2);
}
```

Como se ve, se usa la instrucción **`return`** para indicar cuál es el valor que ha de devolver el método. Este valor es el resultado de ejecutar la expresión `(par1 + par2);` es decir, es la suma de los valores pasados a sus parámetros par1 y par2 al llamarlo.

7.7.2. Parámetros de salida

Un **parámetro de salida** se diferencia de uno de entrada en que todo cambio que se le realice en el código del método al que pertenece afectará al objeto que se le pase al llamar dicho método tanto si éste es de un tipo por valor como si es de un tipo referencia. Esto se debe a que lo que a estos parámetros se les pasa es siempre una referencia al valor que almacenaría una variable del tipo del objeto que se les pase.

Cualquier parámetro de salida de un método **siempre ha de modificarse** dentro del cuerpo del método y además dicha modificación ha de hacerse antes que cualquier lectura de su valor. Si esto no se hiciese así el compilador lo detectaría e informaría de ello con un error. Por esta razón

Ing. Fausto Rocchi

es posible pasar parámetros de salida que sean variables no inicializadas, pues se garantiza que en el método se inicializarán antes de leerlas. Además, tras la llamada a un método se considera que las variables que se le pasaron como parámetros de salida ya estarán inicializadas, pues dentro del método seguro que se las inicializa.

Nótese que este tipo de parámetros permiten diseñar métodos que devuelvan múltiples objetos: un objeto se devolvería como valor de retorno y los demás se devolverían a través de los parámetros de salida.

Los parámetros de salida se definen de forma parecida a los parámetros de entrada pero se les ha de añadir la palabra reservada **out**. Es decir, se definen así:

```
out <tipoParámetro> <nombreParámetro>
```

Al llamar a un método que tome parámetros de este tipo también se ha preceder el valor especificado para estos parámetros del modificador **out**. Una utilidad de esto es facilitar la legibilidad de las llamadas a métodos. Por ejemplo, dada una llamada de la forma:

```
a.f(x, out z);
```

Es fácil determinar que lo que se hace es llamar al método f() del objeto a pasándole x como parámetro de entrada y z como parámetro de salida. Además, también se puede deducir que el valor de z cambiará tras la llamada.

Sin embargo, la verdadera utilidad de forzar a explicitar en las llamadas el tipo de paso de cada parámetro es que permite evitar errores derivados de que un programador pase una variable a un método y no sepa que el método la puede modificar. Teniéndola que explicitar se asegura que el programador sea consciente de lo que hace.

7.7.3. Parámetros por referencia

Un **parámetro por referencia** es similar a un parámetro de salida sólo que no es obligatorio modificarlo dentro del método al que pertenece, por lo que será obligatorio pasarle una variable inicializada ya que **no se garantiza su inicialización en el método**.

Los parámetros por referencia se definen igual que los parámetros de salida pero sustituyendo el modificador **out** por el modificador **ref**. Del mismo modo, al pasar valores a parámetros por referencia también hay que precederlos del **ref**.

7.8. Constructores

Los **constructores** de un tipo de datos son métodos especiales que se definen como miembros de éste y que contienen código a ejecutar cada vez que se cree un objeto de ese tipo. Éste código suele usarse para labores de inicialización de los campos del objeto a crear, sobre todo cuando el valor de éstos no es constante o incluye acciones más allá de una asignación de valor (aperturas de ficheros, accesos a redes, etc.).

Hay que tener en cuenta que la ejecución del constructor siempre se realiza después de haberse inicializado todos los campos del objeto, ya sea con los valores iniciales que se hubiesen especificado en su definición o dejándolos con el valor por defecto de su tipo.

Aparte de su especial sintaxis de definición, los constructores y los métodos normales tienen una diferencia muy importante: **los constructores no se heredan**.

7.8.1. Definición de constructores

La sintaxis básica de definición de constructores consiste en definirlos como cualquier otro método pero dándoles el mismo nombre que el tipo de dato al que pertenecen y no indicando el tipo de valor de retorno debido a que nunca pueden devolver nada. Es decir, se usa la sintaxis:

```
<modificadores> <nombreTipo>(<parámetros>)
{
  <código>
}
```

Ing. Fausto Locchi

Un constructor **nunca puede devolver ningún tipo de objeto** porque, como ya se ha visto, sólo se usa junto al operador **new**, que devuelve una referencia al objeto recién creado. Por ello, es absurdo que devuelva algún valor ya que nunca podría ser capturado en tanto que **new** nunca lo devolvería. Por esta razón el compilador considera erróneo indicar algún tipo de retorno en su definición, incluso aunque se indique **void**.

7.8.2. Llamada al constructor

Al constructor de una clase se le llama en el momento en que se crea algún objeto de la misma usando el operador **new**. De hecho, la forma de uso de este operador es:

```
new <llamadaConstructor>
```

Por ejemplo, el siguiente programa demuestra cómo al crearse un objeto se ejecuta las instrucciones de su constructor:

```
class Prueba
{
  Prueba(int x)
  {
    System.Console.Write("Creado objeto Prueba con x={0}", x);
  }

  public static void Main()
  {
    Prueba p = new Prueba(5);
  }
}
```

La salida por pantalla de este programa demuestra que se ha llamado al constructor del objeto de clase Prueba creado en Main(), pues es:

```
Creado objeto Prueba con x=5;
```

7.8.3. Llamadas entre constructores

Al igual que ocurre con cualquier otro método, también es posible sobrecargar los constructores. Es decir, se pueden definir varios constructores siempre y cuando éstos tomen diferentes números o tipos de parámetros. Además, desde el código de un constructor puede llamarse a otros constructores del mismo tipo de dato antes de ejecutar las instrucciones del cuerpo del primero. Para ello se añade un **inicializador this** al constructor, que es estructura que precede a la llave de apertura de su cuerpo tal y como se muestra en el siguiente ejemplo:

```
class A {
  int total;

  A(int valor): this(valor, 2);  { } // (1)

  A(int valor, int peso) { // (2)
    total = valor * peso;
  }
}
```

El **this** incluido hace que la llamada al constructor (1) de la clase A provoque una llamada al constructor (2) de esa misma clase en la que se le pase como primer parámetro el valor originalmente pasado al constructor (1) y como segundo parámetro el valor 2. Es importante señalar que la llamada al constructor (2) en (1) se hace **antes de ejecutar** cualquier instrucción de (1).

Del mismo modo que en la definición de un constructor de un tipo de datos es posible llamar a otros constructores del mismo tipo de datos, también es posible hacer llamadas a constructores de su tipo padre sustituyendo en su inicializador la palabra reservada **this** por **base**. Por ejemplo:

```
class A
{
  int total;

  A(int valor, int peso)
  {
    total = valor * peso;
  }
}

class B : A
{
  B(int valor) : base(valor, 2) { }
}
```

Ing. Fausto Iocchi

En ambos casos, los valores pasados como parámetros en el inicializador no pueden contener referencias a campos del objeto que se esté creando, ya que se considera que un objeto no está creado hasta que no se ejecute su constructor y, por tanto, al llamar al inicializador aún no está creado.

7.8.4. Constructor por defecto

Todo tipo de datos ha de disponer de al menos un constructor. Cuando se define un tipo sin especificar ninguno el compilador considera que implícitamente se ha definido uno sin cuerpo ni parámetros de la siguiente forma:

```
public <nombreClase>() : base()
{
}
```

En el momento en se defina explícitamente algún constructor el compilador dejará de introducir implícitamente el anterior. Hay que tener especial cuidado con la llamada que este constructor por defecto realiza en su inicializador, pues pueden producirse errores como el del siguiente ejemplo:

```
class A
{
  public A(int x)
  {
  }
}

class B: A
{
  public static void Main()
  {
    B b = new B();  // Error: No hay constructor base
  }
}
```

En este caso, la creación del objeto de clase B en Main() no es posible debido a que el constructor que por defecto el compilador crea para la clase B llama al constructor sin parámetros de su clase base A, pero A carece de dicho constructor porque no se le ha definido explícitamente ninguno con esas características pero se le ha definido otro que ha hecho que el compilador no le defina implícitamente el primero.

Otro error que podría darse consistiría en que aunque el tipo padre tuviese un constructor sin parámetros, éste fuese privado y por tanto inaccesible para el tipo hijo.

También es importante señalar que aún en el caso de que definamos nuestras propios constructores, si no especificamos un inicializador el compilador introducirá por nosotros uno de la forma:base() Por tanto, en estos casos también hay que asegurarse de que el tipo donde se haya definido el constructor herede de otro que tenga un constructor sin parámetros no privado.

7.9. Destructores

Al igual que es posible definir métodos constructores que incluyan código que gestione la creación de objetos de un tipo de dato, también es posible definir un **destructor** que gestione cómo se destruyen los objetos de ese tipo de dato. Este método suele ser útil para liberar recursos tales como los ficheros o las conexiones de redes abiertas que el objeto a destruir estuviese acaparando en el momento en que se fuese a destruir.

La destrucción de un objeto es realizada por el recolector de basura cuando realiza una recolección de basura y detecta que no existen referencias a ese objeto ni en pila, ni en registros ni desde otros objetos sí referenciados. Las recolecciones se inician automáticamente cuando el recolector detecta que queda poca memoria libre o que se va a finalizar la ejecución de la aplicación, aunque también puede forzarse llamando al método **Collect()** de la clase **System.GC**.

La sintaxis que se usa para definir un destructor es la siguiente:

```
~<nombreTipo>()
{
  <código>
}
```

Ing. Fausto Iacchi

Tras la ejecución del destructor de un objeto de un determinado tipo **siempre** se llama al destructor de su tipo padre, formándose así una cadena de llamadas a destructores que acaba al llegarse al destructor de **object**. Éste último destructor no contiene código alguno, y dado que **object** no tiene padre, tampoco llama a ningún otro destructor.

Los destructores no se heredan. Sin embargo, para asegurar que la cadena de llamadas a destructores funcione correctamente si no incluimos ninguna definición de destructor en un tipo, el compilador introducirá en esos casos una por nosotros de la siguiente forma:

```
~<nombreTipo>()
{
}
```

El siguiente ejemplo muestra como se definen destructores y cómo funciona la cadena de llamada a destructores:

```
using System;

class A
{
  ~A()
  {
    Console.WriteLine("Destruido objeto de clase A");
  }
}

class B : A
{
  ~B()
  {
    Console.WriteLine("Destruido objeto de clase B");
  }

  public static void Main()
  {
    new B();
  }
}
```

El código del método Main() de este programa crea un objeto de clase B pero no almacena ninguna referencia al mismo. Luego finaliza la ejecución del programa, lo que provoca la actuación del recolector de basura y la destrucción del objeto creado llamando antes a su destructor. La salida que ofrece por pantalla el programa demuestra que tras llamar al destructor de B se llama al de su clase padre, ya que es:

```
Destruido objeto de clase B
Destruido objeto de clase A
```

Nótese que aunque no se haya guardado ninguna referencia al objeto de tipo B creado y por tanto sea inaccesible para el programador, al recolector de basura no le pasa lo mismo y siempre tiene acceso a los objetos, aunque sean inútiles para el programador.

Es importante recalcar que no es válido incluir ningún modificador en la definición de un destructor, ni siquiera modificadores de acceso, ya que como nunca se le puede llamar explícitamente no tiene ningún nivel de acceso para el programador.

7.10. Propiedades

7.10.1. Concepto de propiedad

Una **propiedad** es una mezcla entre el concepto de campo y el concepto de método. Externamente es accedida como si de un campo normal se tratase, pero internamente es posible asociar código a ejecutar en cada asignación o lectura de su valor. Éste código puede usarse para comprobar que no se asignen valores inválidos, para calcular su valor sólo al solicitar su lectura, etc.

Una propiedad no almacena datos, sino sólo se utiliza como si los almacenase. En la práctica lo que se suele hacer escribir como código a ejecutar cuando se le asigne un valor, código que controle que ese valor sea correcto y que lo almacene en un campo privado si lo es; y como código a ejecutar cuando se lea su valor, código que devuelva el valor almacenado en ese campo público. Así se simula que se tiene un campo público sin los inconvenientes que estos presentan por no poderse

Ing. Fausto Pucci

controlar el acceso a ellos.

7.10.2. **Definición de propiedades**

Para definir una propiedad se usa la siguiente sintaxis:

```
<tipoPropiedad> <nombrePropiedad>
{
  set
  {
    <códigoEscritura>
  }
  get
  {
    <códigoLectura>
  }
}
```

Una propiedad así definida sería accedida como si de un campo de tipo <tipoPropiedad> se tratase, pero en cada lectura de su valor se ejecutaría el <códigoLectura> y en cada escritura de un valor en ella se ejecutaría <códigoEscritura>.

Al escribir los bloques de código **get** y **set** hay que tener en cuenta que dentro del código **set** se puede hacer referencia al valor que se solicita asignar a través de un parámetro especial del mismo tipo de dato que la propiedad llamado **value** (luego nosotros no podemos definir uno con ese nombre en <códigoEscritura>); y que dentro del código **get** se ha de devolver siempre un objeto del tipo de dato de la propiedad.

En realidad el orden en que aparezcan los bloques de código **set** y **get** es irrelevante. Además, es posible definir propiedades que sólo tengan el bloque **get** (**propiedades de sólo lectura**) o que sólo tengan el bloque **set** (**propiedades de sólo escritura**) Lo que no es válido es definir propiedades que no incluyan ninguno de los dos bloques.

Nótese que, aunque en el siguiente ejemplo se ha optado por asociar un campo privado valor a la propiedad `PropiedadEjemplo`, en realidad nada obliga a que ello se haga y es posible definir propiedades que no tengan campos asociados. Es decir, una propiedad no se tiene porqué corresponder con un almacén de datos (una variable en particular).

```
using System;

class B
{
  private int valor;

  public int PropiedadEjemplo
  {
    get
    {
      Console.WriteLine("Leído {0} de PropiedadEjemplo", valor);
      return valor;
    }
    set
    {
      valor = value;
      Console.WriteLine("Escrito {0} en PropiedadEjemplo", valor);
    }
  }
}
```

7.10.3. **Acceso a propiedades**

La forma de acceder a una propiedad, ya sea para lectura o escritura, es exactamente la misma que la que se usaría para acceder a un campo de su mismo tipo. Por ejemplo, se podría acceder a la propiedad de un objeto de la clase B del ejemplo anterior con:

```
B obj = new B();

obj.PropiedadEjemplo++;
```

El resultado que por pantalla se mostraría al hacer una asignación como la anterior sería:

```
Leído 0 de PropiedadEjemplo;
Escrito 1 en PropiedadEjemplo;
```

Nótese que en el primer mensaje se muestra que el valor leído es 0 porque

Ing. Fausto Iocchi

lo que devuelve el bloque **get** de la propiedad es el valor por defecto del campo privado valor, que como es de tipo **int** tiene como valor por defecto el cero.

7.11. Herencia

7.11.1. Concepto de herencia

El mecanismo de **herencia** es uno de los pilares fundamentales en los que se basa la programación orientada a objetos. Es un mecanismo que permite definir nuevas clases a partir de otras ya definidas de modo que si en la definición de una clase indicamos que ésta deriva de otra, entonces la primera a la que se le suele llamar **clase hija** será tratada por el compilador automáticamente como si su definición incluyese la definición de la segunda a la que se le suele llamar **clase padre** o **clase base**. Las clases que derivan de otras se definen usando la siguiente sintaxis:

```
class <nombreHija>:<nombrePadre>
{
  <miembrosHija>
}
```

A los miembros definidos en <miembrosHijas> se les añadirán los que hubiésemos definido en la clase padre. Por ejemplo, a partir de la clase Persona puede crearse una clase Trabajador así:

```
class Trabajador: Persona
{
  public int Sueldo;

  public Trabajador(string nombre, int edad, int ced, int sueldo)
                   :base(nombre, edad, ced)
  {
    Sueldo = sueldo;
  }
}
```

Los objetos de esta clase Trabajador contarán con los mismos miembros que los objetos Persona y además incorporarán un nuevo campo llamado Sueldo que almacenará el dinero que cada trabajador gane. Nótese además que a la hora de escribir el constructor de esta clase ha sido necesario escribirlo con una sintaxis especial consistente en preceder la llave de apertura del cuerpo del método de una estructura de la forma:

```
: base(<parametrosBase>)
```

A esta estructura se le llama **inicializador base** y se utiliza para indicar cómo deseamos inicializar los campos heredados de la clase padre. No es más que una llamada al constructor de la misma con los parámetros adecuados, y si no se incluye el compilador consideraría por defecto que vale**:base()**, lo que sería incorrecto en este ejemplo debido a que Persona carece de constructor sin parámetros.

Un ejemplo que pone de manifiesto cómo funciona la herencia es mostrado a continuación. Nótese que en el mismo ha sido necesario prefijar la definición de los miembros de Persona de la palabra reservada **public**. Esto se debe a que por defecto los miembros de un tipo sólo son accesibles desde código incluido dentro de la definición de dicho tipo, e incluyendo **public** conseguimos que sean accesibles desde cualquier código, como el método Main() definido en Trabajador. **public** es lo que se denomina un **modificador de acceso** (los cuales ya hemos visto en apartados anteriores).

```
using System;

class Persona
{
  public string Nombre;   // Campo que almacena su nombre
  public int Edad;        // Campo que almacena su edad
  public int Cedula;      // Campo que almacena su cédula

  void Cumpleaños()   // Incrementa en uno de edad del objeto Persona
  {
    Edad++;
  }

  // Constructor de Persona
  public Persona(string nombre, int edad, int cedula) {
    Nombre = nombre;
    Edad = edad;
    Cedula = cedula;
  }
}
```

→

Ing. Fausto Iocchi

```
class Trabajador : Persona
{
  public int Sueldo; // Campo que almacena cuánto gana

  Trabajador(string nombre, int edad, int ced, int sueldo)
            :base(nombre, edad, ced)
  {
    // Inicializamos cada Trabajador en base al constructor de Persona
    Sueldo = sueldo;
  }

  public static void Main()
  {
    Trabajador p = new Trabajador("Josan", 22, 77588260, 100000);

    Console.WriteLine ("Nombre=" + p.Nombre);
    Console.WriteLine ("Edad=" + p.Edad);
    Console.WriteLine ("Cedula=" + p.Cedula);
    Console.WriteLine ("Sueldo=" + p.Sueldo);
  }
}
```

7.11.2. **Llamadas por defecto al constructor base**

Si en la definición del constructor de alguna clase que derive de otra no incluimos inicializador base el compilador considerará que éste es **:base()** Por ello hay que estar seguros de que si no se incluye **base** en la definición de algún constructor, el tipo padre del tipo al que pertenezca disponga de constructor sin parámetros.

Es especialmente significativo reseñar el caso de que no demos la definición de ningún constructor en la clase hija, ya que en estos casos la definición del constructor que por defecto introducirá el compilador será en realidad de la forma:

```
<nombreClase>(): base()
{
}
```

Es decir, este constructor siempre llama al constructor sin parámetros del padre del tipo que estemos definiendo, y si ése no dispone de alguno se producirá un error al compilar.

7.12. Estructuras

7.12.1. **Concepto de estructura**

Una **estructura** es un tipo especial de clase pensada para representar objetos ligeros. Es decir, que ocupen poca memoria y deban ser manipulados con velocidad, como objetos que representen puntos, fechas, etc. Ejemplos de estructuras son la mayoría de los tipos básicos (excepto **string** y **object**), y de hecho las estructuras junto con la redefinición de operadores son la forma ideal de definir nuevos tipos básicos a los que se apliquen las mismas optimizaciones que a los predefinidos.

7.12.2. **Diferencias entre clases y estructuras**

A diferencia de una clase y fiel a su espíritu de "ligereza", una estructura no puede derivar de ningún tipo y ningún tipo puede derivar de ella. Por estas razones sus miembros no pueden incluir modificadores relativos a herencia. Otra diferencia entre las estructuras y las clases es que sus variables no almacenan referencias a zonas de memoria dinámica donde se encuentran almacenados objetos sino directamente referencian a objetos. Por ello se dice que las clases son **tipos referencia** y las estructuras son **tipos valor**.

Una primera consecuencia de esto es que los accesos a miembros de objetos de tipos valor son mucho más rápidos que los accesos a miembros de pilas, ya que es necesario pasar por una referencia menos a la hora de acceder a ellos. Además, el tiempo de creación y destrucción de estructuras también es inferior. De hecho, la destrucción de los objetos almacenados en pila es prácticamente inapreciable ya que se realiza con un simple decremento del puntero de pila y no interviene en ella el recolector de basura.

Otra consecuencia de lo anterior es que cuando se realicen asignaciones entre variables de tipos valor, lo que se va a copiar en la variable destino es el objeto almacenado por la variable fuente y no la dirección de memoria dinámica a la que apuntaba ésta. Por ejemplo, dado el siguiente

Ing. Fausto Ioschi

tipo (nótese que las estructuras se definen igual que las clases pero usando la palabra reservada **struct** en vez de **class**):

```
struct Point
{
  public int x, y;

  public Point(int x, int y)
  {
    this.x = x;
    this.y = y;
  }
}
```

Si usamos este tipo en un código como el siguiente:

```
Punto p = new Punto(10, 10);
Punto p2 = p;

p2.x = 100;
Console.WriteLine(p.x);
```

Lo que se mostrará por pantalla será 10. Esto se debe a que el valor de x modificado es el de p2, que como es una copia de p los cambios que se le hagan no afectarán a p. Sin embargo, si Punto hubiese sido definido como una clase entonces sí que se hubiese mostrado por pantalla 100, ya que en ese caso lo que se habría copiado en p2 habría sido una referencia a la misma dirección de memoria dinámica referenciada por p, por lo que cualquier cambio que se haga en esa zona a través de p2 también afectará a p.

De lo anterior se deduce que la asignación entre objetos de tipos estructuras es mucho más lenta que la asignación entre objetos de clases, ya que se ha de copiar un objeto completo y no solo una referencia. Para aliviar esto al pasar objetos de tipos estructura como parámetros, se da la posibilidad de pasarlos como parámetros por referencia (modificador **ref**) o parámetros de salida (**out**) en vez de como parámetros de entrada.

7.12.3. **Constructores**

La diferencia más importante entre los constructores de ambos tipos se encuentra en la implementación del constructor sin parámetros: como los objetos estructura no pueden almacenar el valor por defecto **null** cuando se declaran sin usar constructor ya que ese valor indica referencia a posición de memoria dinámica indeterminada y los objetos estructura no almacenan referencias, toda estructura siempre tiene definido un constructor sin parámetros que lo que hace es darle en esos casos un valor por defecto a los objetos declarados. Ese valor consiste en poner a cero toda la memoria ocupada por el objeto, lo que tiene el efecto de dar como valor a cada campo el cero de su tipo. Por ejemplo, el siguiente código imprime un 0 en pantalla:

```
Punto p = new Punto();

Console.WriteLine(p.X);
```

Y el siguiente también:

```
using System;

struct Punto {
  public int X, Y;
}

class EjemploConstructorDefecto {
  Punto p;

  public static void Main() {
    Console.WriteLine(p.X);
  }
}
```

Sin embargo, el hecho de que este constructor por defecto se aplique no implica que se pueda acceder a las variables locales sin antes inicializarlas con otro valor. Por ejemplo, el siguiente fragmento de código de un método sería incorrecto:

```
Punto p;

Console.WriteLine(p.X);  // X no inicializada
```

Sin embrago, como a las estructuras declaradas sin constructor no se les da el valor por defecto **null**, sí que sería válido:

Ing. Fausto Iocchi

```
Punto p;

p.X = 2;
Console.WriteLine(p.X);
```

Para asegurar un valor por defecto común a todos los objetos estructura, se prohíbe a los programadores darles su propia definición del constructor sin parámetros. Mientras que en las clases es opcional implementarlo y si no se hace el compilador introduce uno por defecto, en las estructuras no es válido hacerlo. Además, aún en el caso de que se definan otros constructores, el constructor sin parámetros seguirá siendo introducido automáticamente por el compilador a diferencia de cómo ocurría con las clases donde en ese caso el compilador no lo introducía.

Por otro lado, para conseguir que el valor por defecto de todos los objetos estructuras sea el mismo, se prohíbe darles una valor inicial a sus campos en el momento de declararlos, pues si no el constructor por defecto habría de tenerlos en cuenta y su ejecución sería más ineficiente. Por esta razón, los constructores definidos por el programador para una estructura han de inicializar todos sus miembros no estáticos en tanto que antes de llamarlos no se les da ningún valor inicial.

Ing. Fausto Iocchi

Unidad VIII

Objetivo:

Al finalizar esta unidad, el participante tendrá una visión más amplia de lo que es hacer un programa bajo la perspectiva de la programación orientada a objetos, lo cual apoyará su aprendizaje en el desarrollo de programas.

Contenido:

8. Ejercicio completo usando POO.
   8.1. Creación de la clase principal y el método de inicio.
   8.2. Uso de la instrucción condicional *if*.
   8.3. Uso de operaciones aritméticas.
   8.4. Uso de métodos creados por el usuario.
   8.5. Creación y uso de objetos.
   8.6. Creación de propiedades y uso de constructores.
   8.7. Uso de ciclos iterativos: *for*.
   8.8. Apoyo de otra clase y uso del ciclo *do-while*.
   8.9. Herencia y uso del objeto especial *base*.

Actividades del participante:

- Realizar lectura previa sobre la unidad anterior.
- Discutir en grupo.

Evaluación multidireccional:

- Evaluación formativa: a través de las discusiones de los conocimientos e intervenciones en clase.

Ejercicio Completo usando POO

8. Ejercicio

Hagamos a continuación un ejercicio, aplicando todos los conceptos de POO aprendidos. Para esto, vamos a ir desarrollando el problema que se plantea en varias fases, para irle incorporando paso a paso todos los elementos aprendidos a los largo de esta guía. Posteriormente, en el siguiente capítulo, se resolverán otros problemas, pero estos ya sin la explicación detallada que se le dará a este, ya que se asume que usted es capaz de entenderlo y aplicarlo a otros problemas que se le presenten en la realidad. Luego, en la última unidad encontrará un gran número de problemas propuestos, los cuales usted debería hacerlos, para que complemente sus conocimientos adquiridos.

Todos los problemas resueltos en la presente guía, como ya se dijo en capítulos anteriores, estarán basados en aplicaciones de tipo Consola, por lo que se estará usando la entrada y la salida estándar.

8.1. Creación de la clase principal y el método de inicio

En este primer problema abarcaremos lo que es la declaración de una clase principal, el método de entrada a la aplicación (el método especial Main) y la asignación de valores primitivos a variables declaradas dentro del programa. El problema a resolver es el siguiente: Escribir un programa, utilizando para esto el lenguaje de programación C#, en el cual se lea el ancho y el alto de un triángulo:

```
// empezamos por incorporar la librería estándar de .NET
usign System;

// ahora, declaramos lo que será nuestra clase principal
class CPrincipal {
  // el punto de entrada para nuestra aplicación será el método Main
  public static void Main(/* no importa el argumento es opcional */) {
    // declaramos dos variables de tipo double, en las cuales se
    // almacenarán tanto el alto como el ancho de nuestro triángulo
    double alto;
                                                                    →
```

Ing. Fausto Lorchi

```
    double ancho;
    // nótese que son de tipo double ya que estos valores podrían
    // contener números con posiciones decimales

    // escribimos un literal de string en el cual se avisa al usuario
    // que se va a leer el alto
    Console.Write("Ingrese el alto: ");

    // se procede a leer, desde la consola, el valor de la variable
    alto = double.Parse(Console.ReadLine());

    // hacer lo mismo ahora con la variable de ancho
    Console.Write("Ingrese el ancho: ");
    ancho = double.Parse(Console.ReadLine());
  }
}
```

Como se ve, fue sencillamente fácil hacer esto. Se ve entonces cómo se utilizó la operación de asignación para obtener y guardar los valores introducidos por el usuario a través de lo que escriba por el teclado (siendo este la entrada estándar de un computador). Ahora, ¿qué tal si ahora le empezamos a colocar restricciones al programa?

8.2. Uso de la instrucción condicional *if*

Tomemos el problema anterior, pero ahora deberemos validar que ambos valores (ancho y alto) deban ser números positivos, ya que no es lógico que las medidas de un triángulo (o figura cualquiera) sean 0 o negativas. En caso de que alguno (o ambos) valores sean negativos, se desplegará un mensaje de error indicándolo.

```
usign System;

class CPrincipal {
  public static void Main() {
    double alto;
    double ancho;

    Console.Write("Ingrese el alto: ");
    alto = double.Parse(Console.ReadLine());

    Console.Write("Ingrese el ancho: ");
    ancho = double.Parse(Console.ReadLine());
```

```
    // validemos que sean positivos ambos
    if ((ancho > 0.0) && (alto > 0.0)) {
      // hacer algo acá… aún no
    } else {
      // uno de los dos (o ambos) valores son negativos
      // veamos cual fue para indicárselo al usuario

      // probamos primero el alto
      if (alto <= 0.0) {
        Console.WriteLine("el valor ALTO no puede ser negativo");
      }

      // probamos ahora el ancho
      if (ancho <= 0.0) {
        Console.WriteLine("el valor ANCHO no puede ser negativo");
      }
    }
  }
}
```

Ahora ya tenemos sabemos cómo validar datos… correcto!!! Utilizando la instrucción condicional **if**, conjuntamente con una operación lógica y operadores relacionales. Una vez que ya tengamos esto claro, podremos continuar a la siguiente fase, que será calcular y mostrar al usuario el área de dicha figura geométrica, que en este caso es representada por el triángulo.

8.3. Uso de operaciones aritméticas

A este punto, el programa en sí no hace nada. Tan solo lee un par de valores, los almacena en memoria a través de dos variables y realiza unas validaciones, pero no realiza ningún proceso de los datos introducidos por el usuario.

En este paso, vamos a agregarle una funcionalidad, que será la de calcular el área de la figura en cuestión. Posteriormente a esto se deberá mostrar al usuario el cálculo realizado.

Como ya sabemos, el área de un triángulo viene dada por:

$$A = \frac{base \times altura}{2}$$

Ing. Fausto Iocchi

Ahora, para incorporar esto a nuestro programa, debemos agregar una nueva variable, que será la encargada de almacenar este cálculo en la memoria.

```
usign System;

class CPrincipal
{
  public static void Main()
  {
    double alto;
    double ancho;
    double area;  // nuestra variable para el cálculo del área

    Console.Write("Ingrese el alto: ");
    alto = double.Parse(Console.ReadLine());

    Console.Write("Ingrese el ancho: ");
    ancho = double.Parse(Console.ReadLine());

    if ((ancho > 0.0) && (alto > 0.0))
    {
      // todo OK… hacemos el cálculo del área
      area = (alto * ancho) / 2.0;

      // mostrar resultado por cónsola
      Console.WriteLine("Área del triángulo: " + area.ToString());
    }
    else
    {
      if (alto <= 0.0)
      {
        Console.WriteLine("el valor ALTO no puede ser negativo");
      }
      if (ancho <= 0.0)
      {
        Console.WriteLine("el valor ANCHO no puede ser negativo");
      }
    }
  }
}
```

Acá, como se puede observar, hacemos uso de una de las características del lenguaje C#. Esto es cuando mostramos el resultado al usuario, hacemos una suma de cadenas. Sumamos el literal *"Área del triángulo: "* conjuntamente con el valor, que es *double*, de la variable *area*. Esta variable, al ser un objeto que hereda por defecto de la clase *Object*, posee un método llamado *ToString()*, que es la encargada de convertir el valor que tenga esta variable en una cadena.

8.4. Uso de métodos creados por el usuario

Llegado a este punto, nuestro programa ya hace algo… calcula el área de un triángulo. Llegó el momento de separar las funcionalidades de nuestro programa, porque como podemos ver, nuestro programa tiene un comportamiento "lineal", sin ningún subproceso (o métodos) propios de nuestra clase.

Para hacer la "separación", podemos empezar por la entrada de datos. Si observamos bien, en la entrada de ambas variables se repiten los pasos a seguir: mostrar un literal por la consola para luego hacer una lectura de una variable de tipo *double*. Hagamos entonces un método en el cual se reciba un argumento de tipo *string* (que vendría siendo el literal a mostrarle al usuario) y que retorne el valor que el usuario ingrese para el mismo.

Pero antes de hacerlo, debemos tener presentes un par de cosas. Primero, que dicho método que hagamos deberá ser de tipo *static*. ¿Por qué? Ya que como a este método se le va a "llamar" desde nuestro método principal Main (el cual es *static*), este también deberá ser del mismo tipo. Recordemos que desde un método *static* no podremos tener acceso a otro método (aún cuando dichos métodos se encuentren definidos en la misma clase), que no sea de este tipo.

Lo segundo a tener en cuenta es que este nuevo método habrá de ser también de alcance público, o sea, de tipo *public*. Existe una manera de evitar todo esto, y es replanteando la clase desde otro punto de vista, el cual estaremos haciéndolo más adelante, en este mismo capítulo.

Este nuevo método, como ya se dijo, deberá devolver un valor de tipo *double*, para que de este manera poderle asignar, desde donde se llame, el valor retornado, siendo este el ingresado por el usuario.

Como funcionalidad adicional, podremos hacer que este nuevo método realice la validación del valor ingresado por el usuario, indicándole cuando

Ing. Fausto Locchi

exista un error en el valor ingresado (en caso de que este sea un número negativo).

```
usign System;

class CPrincipal
{
  public static void Main() {
    double alto;
    double ancho;
    double area;

    // hacemos uso de nuestro nuevo método
    alto = Lee("alto");
    ancho = Lee("ancho");

    if ((ancho > 0.0) && (alto > 0.0)) {
      area = (alto * ancho) / 2.0;

      Console.WriteLine("Área del triángulo: " + area.ToString());
    }
    // la cláusula else ya no hace falta, ya que el mensaje de error
    // lo mostramos en el nuevo método
  }

  // creamos nuestro nuevo método acá… este método recibirá una cadena
  // como entrada, la cual será mostrada por consola al usuario, leerá
  // también por la consola un valor numérico ingresado por el usuario
  // y se devolverá dicho valor al mismo punto en donde este método
  // sea invocado
  public static double Lee(string var) {
    double valor;

    Console.Write("Ingrese el " + var + ": ");
    valor = double.Parse(Console.ReadLine());

    if (valor <= 0.0) {
      Console.WriteLine("El valor " + var + " no puede ser negativo");
    }
    return valor;
  }
}
```

Bien!!! Ya tenemos un programa que lee un par de variables y realiza el cálculo pertinente al caso. Pero ¿qué pasaría si quisiéramos continuar haciendo otra serie de cálculos? Por ejemplo, calcularle el perímetro del mismo y otras tantas cosas más. Tendríamos que declarar tantas variables dentro de nuestro método Main, a tal punto que se perdería la perspectiva del mismo.

8.5. Creación y uso de objetos

Lo mejor que podremos hacer en casos como lo explicado en al final del punto anterior es ir separando funcionalidades, ya no por métodos, sino por unidades funcionales inherentes o comunes a cada caso. Por ejemplo, podríamos comenzar por separar lo que es el manejo abstracto de un triángulo de la clase principal. De este modo tendremos en nuestro caso dos clases: una, que representaría a nuestro triángulo y la otra nuestra clase principal, en la que estaremos probando desde allí a nuestra nueva clase desarrollada.

Hagamos pues entonces una separación de funcionalidades… "Divide y vencerás" reza una máxima antigua… y dicha máxima no se podría aplicar mejor al caso de la Programación Orientada a Objetos, que mientras más separadas (en un orden lógico funcional, por supuesto) estén las cosas, más fácil será su comprensión y el mantenimiento del mismo.

```
usign System;

// nuestra nueva clase que representará a la figura de un triángulo
class CTriangulo
{
  // recordemos que si queremos acceder a los atributos de una clase,
  // estos deberán ser declarados como públicos
  public double alto;
  public double ancho;
  // el atributo de area lo podremos ignorer debido a que haremos
  // en su lugar un método que nos devuelva dicho valor

  // método público en el cual se calculará el área del triángulo
  public double Area()
  {
    return ((alto * ancho) / 2.0);
  }
}

class CPrincipal
{
  public static void Main()
  {
    // declaramos una variable de tipo CTriangulo y la instaciamos
    // de una vez… todo en un solo paso
    CTriangulo tri = new CTriangulo();

    // ahora, leemos sus valores de alto y ancho, a través de sus
```

→

Ing. Fausto Locchi

```
    // atributos homónimos
    tri.alto = Lee("alto");
    tri.ancho = Lee("ancho");

    if ((tri.ancho > 0.0) && (tri.alto > 0.0))
    {
      // accesamos al método que nos retorna el área
      Console.Write("Área del triángulo: ");
      Console.WriteLine(tri.Area());
    }
  }

  public static double Lee(string var)
  {
    double valor;

    Console.Write("Ingrese el " + var + ": ");
    valor = double.Parse(Console.ReadLine());

    if (valor <= 0.0)
    {
      Console.WriteLine("El valor " + var + " no puede ser negativo");
    }
    return valor;
  }
}
```

Ahora, hemos separado la funcionalidad de la figura de un triángulo en una clase aparte, la hemos instanciado y finalmente utilizado sus atributos y métodos. Pero para poder cumplir con una de las máximas de la POO (la encapsulación), no deberíamos dejar que los atributos se hagan visibles a todo nivel. Para esto, debemos ahora recodificar este programa para convertir dichos atributos en propiedades, y a ser posible, colocarlos de sólo lectura. Si hacemos esto, nos tropezaríamos con un inconveniente: ¿cómo hacemos entonces para colocarles valores a estas propiedades? Solución, en el próximo apartado.

8.6. Creación de propiedades y uso de constructores

Bueno, modifiquemos nuestro programa anterior para que ahora, en vez de utilizar los atributos públicos, accedamos a las características del objeto a través de propiedades, y mantener así el encapsulamiento del mismo.

```
usign System;

class CTriangulo {
  // ahora, los atributos son privados
  double alto;
  double ancho;

  // escribamos código de inicialización de los atributos
  public CTriangulo(double alto, double ancho) {
    this.alto = alto;
    this.ancho = ancho;
  }

  // escribamos código para nuestra propiedad Alto
  public double Alto {
    // será de sólo lectura;
    get {
      return alto;
    }
  }

  // escribamos código para nuestra propiedad Ancho
  public double Ancho {
    // será de sólo lectura;
    get {
      return ancho;
    }
  }

  public double Area() {
    return ((alto * ancho) / 2.0);
  }
}

class CPrincipal {
  public static void Main() {
    // declaremos un par de variables para su uso interno
    double ancho; // variable para el ancho
    double alto;  // variable para el alto

    // declaramos, más aún NO instanciamos la variable tri
    CTriangulo tri;

    // utilizaremos un par de variables "normales" para la lectura
    // de las medidas
    alto = Lee("alto");
    ancho = Lee("ancho");

    if ((ancho > 0.0) && (alto > 0.0)) {
      // ahora SI podremos instanciar el objeto con los
      // valores conocidos
      tri = new CTriangulo(alto, ancho);
```

Ing. Fausto Iocchi

```
      Console.Write("Área del triángulo ");
      Console.Write("de {0}x{1} es: ", tri.Ancho, tri.Alto);
      Console.WriteLine(tri.Area());
    }
  }

  public static double Lee(string var) {
    double valor;

    Console.Write("Ingrese el " + var + ": ");
    valor = double.Parse(Console.ReadLine());

    if (valor <= 0.0) {
      Console.WriteLine("El valor " + var + " no puede ser negativo");
    }
    return valor;
  }
}
```

Ahora vemos como, un simple problema de cálculo de área de un triángulo, se ha convertido en algo "largo y difícil"... pero sabemos que esta es la mejor manera de aplicar la POO, ya que todas las operaciones inherentes a algún objeto que envuelva a la figura geométrica de un triángulo, se hará justamente en una clase, encapsulando sus características, y la interacción con el "mundo exterior" se hará a través de sus métodos.

Bien, ¿qué pasaría si ahora el usuario desea que, en vez de leer y procesar un triángulo, quisiera procesar N triángulos? Según como están las cosas, no le quedaría más remedio, al usuario, que ejecutar nuestra aplicación N veces, lo cual no es muy convincente ni estratégico, y más aún dados los adelantos tecnológicos existentes hoy día (¿para qué tener entonces un computador, si la tarea repetitiva se debe hacer manual?).

Deberemos entonces modificar nuestro programa para cumplir con este nuevo requerimiento del usuario.

8.7. Uso de ciclos iterativos: *for*

Bien, nuestro próximo paso será ahora el de tratar de hacer que el programa realice el proceso N veces. Para esto, debemos saber si la cantidad N es conocida. De ser así, procederemos a leer N y posteriormente, en un ciclo iterativo *for* haremos el proceso.

```
usign System;

class CTriangulo {
  double alto;
  double ancho;

  public CTriangulo(double alto, double ancho) {
    this.alto = alto;
    this.ancho = ancho;
  }

  public double Alto {
    get {
      return alto;
    }
  }

  public double Ancho {
    get {
      return ancho;
    }
  }

  public double Area() {
    return ((alto * ancho) / 2.0);
  }
}

class CPrincipal {
  public static void Main() {
    // usaremos una variable para guardar cuantos triángulos
    // desea procesar el usuario: N
    int N;
    double ancho;
    double alto;

    CTriangulo tri;

    // leemos la cantidad de triángulos que se quiere procesar
    Console.Write("Ingrese N: ");
    N = int.Parse(Console.ReadLine());

    // recurrimos al ciclo for
    for (int i = 0; i < N; i++) {
      Console.WriteLine("Ingrese datos de la figura " + (i + 1));

      alto = Lee("alto");
      ancho = Lee("ancho");

      if ((ancho > 0.0) && (alto > 0.0)) {
```

Ing. Fausto Iocchi

```
        tri = new Ctriangulo(alto, ancho);

        Console.Write("Área del triángulo ");
        Console.Write("de {0}x{1} es: ", tri.Ancho, tri.Alto);
        Console.WriteLine(tri.Area());
      }
    }
  }

  public static double Lee(string var) {
    double valor;

    Console.Write("Ingrese el " + var + ": ");
    valor = double.Parse(Console.ReadLine());

    if (valor <= 0.0) {
      Console.WriteLine("El valor " + var + " no puede ser negativo");
    }
    return valor;
  }
}
```

Resuelto… pero claro, tiene sus fallas. Por ejemplo, ¿qué pasaría si el usuario se equivoca y coloca uno de los datos de algún triángulo negativo? Simplemente que se perdería ese intento de lectura y, si el usuario deseaba procesar 5 triángulos, sólo le quedarán para procesar 4. Pero eso es algo que ya usted debería saber cómo solucionar.

El siguiente paso es ¿cómo haríamos para, en vez de calcular el área de triángulos, también lo podamos hacer para rectángulos? La solución, en el próximo apartado. Los invito a que continúen…

8.8. Apoyo de otra clase y uso del ciclo *do-while*

Acá estaremos ocupado ahora en la implementación de otra nueva clase: la clase para el manejo de la figura geométrica rectangular. Se nos presenta, a primeras de cambio, independientemente de cómo se haga la clase para esta nueva figura, el cómo reconocer que los datos que esté ingresando el usuario sean los de un triángulo o los de un rectángulo.

Para resolver esto, vamos a modificar nuestro programa principal (método Main) para actúe también como un menú, en el que el usuario elegirá qué cuál es la figura que desea procesar (dentro de la lista de N figuras).

Entonces, lo que se hará serán dos cosas: crear una nueva clase denominada en este ejemplo como *TRectangulo* y lo otro que nos queda hacer es el menú del usuario. Para la confección de este menú utilizaremos el ciclo iterativo *do-while*. Veamos cómo sería:

```
using System;

class CTriangulo {
  double alto;
  double ancho;

  public CTriangulo(double alto, double ancho) {
    this.alto = alto;
    this.ancho = ancho;
  }

  public double Alto {
    get {
      return alto;
    }
  }

  public double Ancho {
    get {
      return ancho;
    }
  }

  public double Area() {
    return ((alto * ancho) / 2.0);
  }
}

// he aquí nuestra nueva clase para el manejo de rectángulos… todas
// las propiedades y atributos permanecerán iguales, lo que debe
// cambiar en el método del cálculo del área
class CRectangulo {
  double alto;
  double ancho;

  public CRectangulo(double alto, double ancho) {
    this.alto = alto;
    this.ancho = ancho;
  }

  public double Alto {
    get {
      return alto;
    }
  }
```

Ing. Fausto Iocchi

```
  public double Ancho {
    get {
      return ancho;
    }
  }

  // este método deberá cambiar, ya que el área se calcula de
  // manera diferente
  public double Area() {
    return (alto * ancho);
  }
}

class CPrincipal {
  public static void Main() {
    int N;
    double ancho;
    double alto;

    // nueva variable, que será usada para la opción del menú
    string op;

    CTriangulo tri;
    CRectangulo rec; // nuestra nueva variable para rectángulos

    Console.Write("Ingrese N: ");
    N = int.Parse(Console.ReadLine());

    for (int i = 0; i < N; i++) {
      // ciclo iterativo en el cual se le pedirá al usuario cual es
      // la siguiente figura en procesar
      do {
        Console.WriteLine("1.- Triángulo");
        Console.WriteLine("2.- Rectángulo");
        Console.WriteLine("Escriba su opción y pulse ENTER");
        op = Console.ReadLine();
      } while ((op != "1") && (op != "2"));

      Console.WriteLine("Ingrese datos de la figura " + (i + 1));

      alto = Lee("alto");
      ancho = Lee("ancho");

      if ((ancho > 0.0) && (alto > 0.0)) {
        Console.Write("Área de la figura ");

        if (op == "1") {    // la figura es un triángulo
          tri = new CTriangulo(alto, ancho);
          Console.Write("de {0}x{1} es: ", tri.Ancho, tri.Alto);
          Console.WriteLine(tri.Area());
        } else {            // sino, será un rectángulo
          rec = new CRectangulo(alto, ancho);
```

```
          Console.Write("de {0}x{1} es: ", rec.Ancho, rec.Alto);
          Console.WriteLine(rec.Area());
        }
      }
    }
  }

  public static double Lee(string var) {
    double valor;

    Console.Write("Ingrese el " + var + ": ");
    valor = double.Parse(Console.ReadLine());

    if (valor <= 0.0) {
      Console.WriteLine("El valor " + var + " no puede ser negativo");
    }
    return valor;
  }
}
```

Como se ve, es sumamente sencillo. Lo único que hicimos fue, prácticamente, copiar y pegar la clase *CTriangulo* y nombrarla *CRectangulo*. El otro cambio fue la introducción de un bucle *do-while* para que fungiera como una especie de menú del sistema, en el cual el usuario podrá escoger entre ingresar los datos de un triángulo o los de un rectángulo.

Pero, como se ve (y se dijo), la clase *TRectangulo* no es más que una vulgar copia de la clase *CTriangulo*. Y si vemos bien, entre estas dos clases existe gran similitud. ¿Será que se podrá hacer una clase base, en la que se englobe lo común a estas dos figuras y, que de ella se hereden ambas clases?

8.9. Herencia y uso del objeto especial *base*

Tomemos ahora el código resultante del ítem anterior y modifiquémoslo para implementar la herencia. Debido, y es lógico pensar así, a que ambas clases representan a una figura geométrica en particular, no dejan de ser precisamente eso: una figura geométrica. Si analizamos bien la situación, tenemos que las figuras geométricas que estamos utilizando de ejemplo, tienen mucho en común: ambas tienen medidas de altura y anchura, así

Ing. Fausto Iocchi

como también a ambas se les puede calcular el área ocupada.

Entonces, podremos crear una clase base (TFigura) en la que estarán los atributos comunes a las dos figuras en particular. Posteriormente, hagamos del método de cálculo del área un método que podamos sobrecargar, para que de esta manera poder utilizar a plenitud otro de los conceptos de la POO: el *polimorfismo*.

Otro de los cambios que le podríamos introducir a este programa, para terminar con este tema, es eliminar el ingreso de la cantidad de figuras que el usuario desea procesar y a cambio agregamos una nueva opción a nuestro menú para que el mismo usuario escoja en qué momento desea terminar la ejecución del programa.

```
using System;

// la nueva clase: será la base (padre) de las otras dos clases
class CFigura {
  double alto;
  double ancho;

  // cuando se vaya a heredar una clase de otra, la clase padre DEBERÁ
  // tener definido el constructor por defecto, aún cuando este no
  // haga nada
  public CFigura() {
  }

  public CFigura(double alto, double ancho) {
    this.alto = alto;
    this.ancho = ancho;
  }

  public double Alto {
    get {
      return alto;
    }
  }

  public double Ancho {
    get {
      return ancho;
    }
  }

  // el método de la clase base devolverá SIEMPRE 0, ya que lo ideal
  // es que este método no sea llamado desde ninguna parte...
```

→

←

```
  // por esto es declarada como VIRTUAL, para que desde las clases
  // hijas se pueda sobreescribir
  public virtual double Area() {
    return 0.0;
  }
}

// heredamos ahora de la clase CFigura, por lo que no hacen falta las
// declaraciones de los atributos y propiedades
class CTriangulo : CFigura {
  // hacemos uso del llamado al constructor de la clase padre
  public CTriangulo(double alto, double ancho) : base(alto, ancho) {
  }

  // este método ahora será sobreescrito, gracias a la palabra clave
  // OVERRIDE… y para el cálculo del área, y ya que NO podemos
  // utilizar para ello los atributos (porque son privados),
  // utilizaremos sus propiedades
  public override double Area() {
    return ((Alto * Ancho) / 2.0);
  }
}

// heredamos ahora de la clase CFigura, por lo que no hacen falta las
// declaraciones de los atributos y propiedades... acá se aplica lo
// mismo que para la clase CTriangulo
class CRectangulo : CFigura {
  public CRectangulo(double alto, double ancho) : base(alto, ancho) {
  }

  public override double Area() {
    return (Alto * Ancho);
  }
}

class CPrincipal {
  public static void Main() {
    // ya no necesitaremos más a las variables "N"
    double ancho;
    double alto;
    string op;

    // tampoco requeriremos del uso de variables del tipo CTriangulo o
    // CRectangulo... sólo usaremos CFigura y los demás sale por
    // polimorfismo
    CFigura fig;

    // bucle infinito hasta tanto el usuario no escoja la opción de
    // salida del sistema
    do {
```

→

Ing. Fausto Cocchi

```
      Console.WriteLine("1.- Triángulo");
      Console.WriteLine("2.- Rectángulo");
      Console.WriteLine("9.- Para salir del programa");
      Console.WriteLine("Escriba su opción y pulse ENTER");
      op = Console.ReadLine();

      if ((op == "1") || (op == "2")) {
        Console.WriteLine("Ingrese datos de la figura");

        alto = Lee("alto");
        ancho = Lee("ancho");

        if ((ancho > 0.0) && (alto > 0.0)) {
          // fijarse bien en el uso ahora del objeto "fig"... este
          // objeto es "polimórfico"
          if (op == "1") {
            fig = new CTriangulo(alto, ancho);
          } else {
            fig = new Crectangulo(alto, ancho);
          }
          Console.Write("Área de la figura ");
          Console.Write("de {0}x{1} es: ", fig.Ancho, fig.Alto);
          Console.WriteLine(fig.Area());
        }
      }
    } while (op != "9");
  }

  public static double Lee(string var) {
    double valor;

    Console.Write("Ingrese el " + var + ": ");
    valor = double.Parse(Console.ReadLine());

    if (valor <= 0.0) {
      Console.WriteLine("El valor " + var + " no puede ser negativo");
    }
    return valor;
  }
}
```

Con este ejemplo final hemos puesto de manifiesto los grandes beneficios que nos ofrece la Programación Orientada a Objetos, sobre todo, en la funcionalidad que otorga la herencia de clases. así como del concepto de polimorfismo.

Y bien, ya con esto hemos terminado este largo y gran capítulo. A lo largo de este, se han mostrado todas las facetas de la POO, así como también las instrucciones básicas del lenguaje de programación denominado C#.

Ing. Fausto Iocchi

Unidad IX

Objetivo:

Al finalizar esta unidad, el participante reconocerá muchas de las técnicas utilizadas en la programación orientada a objetos.

Contenido:

9. Ejercicios resueltos.
   9.1. Leer un número entero e indicar si este es par o impar.
   9.2. Leer N números y calcular el promedio de sólo los números impares que se hayan leído.
   9.3. Leer un número entero y determinar si este es primo o no.
   9.4. Leer un número entero positivo N, y calcular la serie $1^1+2^2+3^3+4^4+.....+N^N$
   9.5. Leer una oración e indique cuántas palabras comienzan con una vocal.

Actividades del participante:

- Analizar cada uno de los ejercicios resueltos, encontrándole la lógica funcional a todos ellos.
- Discutir en grupo.

Evaluación multidireccional:

- Evaluación formativa: a través de las discusiones de los conocimientos e intervenciones en clase.

Ejercicios resueltos

9. Ejercicios resueltos

A continuación, veremos una serie de problemas, cada una con una de sus posibles soluciones, utilizando para ello el lenguaje de programación C#.

9.1. Leer un número entero e indicar si este es par o impar.

```
class CNumero
{
  int numero;

  public CNumero(int numero)
  {
    this.numero = numero;
  }

  public int Numero
  {
    get
    {
      return numero;
    }
  }

  public bool EsPar()
  {
    if ((numero % 2) == 0)
    {
      return true;
    }
    return false;
  }
}

class CClase1 {
  static void Main() {
    CNumero num;
    int numero;

    Console.Write("Ingrese un número entero: ");
    numero = int.Parse(Console.ReadLine());
    num = new CNumero(numero);
    Console.Write("El número {0} ", num.Numero);
    Console.Write("es {0}", (num.EsPar() ? "PAR", "IMPAR"));
  }
}
```

Ing. Fausto Iocchi

9.2. Leer N números y calcular el promedio de sólo los números impares que se hayan leído.

```
class CNumero {
  int numero;
  int suma;
  int cont;
  double promedio;

  public CNumero() {
    numero = 0;
    suma = 0;
    cont = 0;
    promedio = 0.0;
  }

  public int Numero {
    set {
      numero = value;

      if (EsPar() == false) {
        cont++;
        suma += numero;
        // hacemos un cast explícito, ya que ambos términos
        // (suma y cont) son enteros y su división será por
        // tanto también entera
        promedio = (double)(suma / cont);
      }
    }
  }

  public int Promedio {
    get {
      return promedio;
    }
  }

  private bool EsPar() {
    if ((numero % 2) == 0) {
      return true;
    }
    return false;
  }
}

class CClase2 {
  static void Main() {
    CNumero num = new CNumero();
    int N;

    Console.Write("Ingrese cantidad de números: ");
```

```
    N = int.Parse(Console.ReadLine());

    for (int i = 0; i < N; i++) {
      Console.Write("Ingrese un número entero {0}: ", (i + 1));
      Num.Numero = int.Parse(Console.ReadLine());
    }
    Console.Write("El promedio de los números ");
    Console.Write("impares es {0}", num.Promedio);
  }
}
```

9.3. Leer un número entero y determinar si este es primo o no.

```
class CNumero
{
  public bool EsPrimo(int num)
  {
    int divisor = 2;

    if (num < 0)
    {
      num = -num;
    }
    if (num == 1)
    {
      return true;
    }
    while (((num % divisor) != 0) && (divisor < num))
    {
      divisor++;
    }

    return (divisor == num);
  }
}

class CClase3
{
  public static void Main()
  {
    int numero;
    CNumero obj = new CNumero();

    Console.Write("Ingrese un número: ");
    numero = int.Parse(Console.ReadLine());

    Console.Write((obj.EsPrimo(numero) ? "" : "No "));
    Console.WriteLine("es primo");
  }
}
```

Ing. Fausto Locchi

9.4. Leer un número entero positivo N, y calcular la serie $1^1+2^2+3^3+4^4+.....+N^N$

```
class CNumero
{
  public int Sumatoria(int num)
  {
    int suma = 0;

    for (int i = 1; i <= num; i++)
    {
      suma += (int)Math.Pow(i, i);
    }
    return suma;
  }
}

class CClase4
{
  public static void Main()
  {
    int numero;
    CNumero obj = new CNumero();

    Console.Write("Ingrese un número: ");
    numero = int.Parse(Console.ReadLine());

    if (numero > 0)
    {
      Console.WriteLine(obj.Sumatoria(numero));
    }
  }
}
```

9.5. Leer una oración e indique cuántas palabras comienzan con una vocal.

```
class COracion
{
                                                    →
```

```
                                                    ←
  static string vocales = "aeiou";

  public int Contar(string oracion) {
    int cont = 0;
    int pos;
    string palabra;
    char letra;

    while (oracion.Length > 0) {
      pos = oracion.IndexOf(" ");

      if (pos >= 0) {
        palabra = oracion.Substring(0, pos);
        oracion = oracion.Remove(0, pos + 1);
      } else {
        palabra = oracion;
        oracion = "";
      }
      letra = palabra.ToLower()[0];

      if (vocales.IndexOf(letra) >= 0) {
        cont++;
      }
    }
    return cont;
  }
}

class CClase5
{
  public static void Main() {
    string cadena;
    COracion obj = new COracion();

    Console.Write("Ingrese una oración: ");
    cadena = Console.ReadLine();
    Console.WriteLine(obj.Contar(cadena));
  }
}
```

Ing. Fausto Locchi

Unidad X

Objetivo:

Al finalizar esta unidad, el participante obtendrá la pericia técnica requerida por la materia, aumentándose así su desarrollo profesional.

Contenido:

10. Problemas propuestos.
    - Proposiciones lógicas.
    - Tablas de verdad.
    - Desarrollo de programas.

Actividades del participante:

- Analizar cada uno de los ejercicios propuestos, dándoles solución satisfactoria a todos ellos.
- Discutir en grupo.

Evaluación multidireccional:

- Evaluación formativa: a través de las discusiones de los conocimientos e intervenciones en clase.

Problemas propuestos

10. Problemas propuestos

Simbolizar completamente las proposiciones siguientes, utilizando los símbolos correspondientes a cada término de enlace. Indicar las proposiciones simples sustituidas por cada letra mayúscula.

10.1. En el hemisferio sur, julio no es un mes de verano.
10.2. Si dos pulsaciones se atraviesan, continúan conservando la forma original.
10.3. O Jaime no es puntual o Tomas llega tarde.
10.4. Ni Antonio ni Ana estudian en la universidad.
10.5. O Pedro es presidente y Juan es tesorero, o Jaime es tesorero.
10.6. Si este cuadro es negro entonces aquel cuadro es rojo y su rey está sobre el cuadro rojo.
10.7. A la vez, si este cuadro es negro, entonces, aquel cuadro es rojo y su rey está sobre el cuadro rojo.
10.8. Patinaremos sí y sólo si el hielo no es demasiado delgado.

Responda a cada pregunta utilizando un razonamiento lógico y/o basado en la tabla de verdad.

10.9. Sean P, Q, R y S fórmulas. Si se sabe únicamente que P es verdadero, ¿Qué puede afirmarse del valor de verdad de cada una las proposiciones siguientes?

$P \wedge Q$	$R \rightarrow P$	$S \rightarrow \neg P$
$R \vee P$	$P \rightarrow Q$	$R \rightarrow (S \rightarrow P)$
$R \wedge P$	$P \rightarrow P \vee S$	$P \vee S \rightarrow (Q \wedge \neg P)$
$S \vee \neg P$	$\neg P \rightarrow Q \wedge R$	$Q \wedge \neg P \rightarrow R \wedge Q$

10.10. ¿Qué puede concluirse de cada una de las proposiciones anteriores, en los siguientes casos?

Ing. Fausto Rocchi

- Si P es falsa.
- Si P es falsa, Q es verdadera y R es verdadera.

10.11. Sean P, Q y R fórmulas, entonces:

- Si R v P → Q ^ P es falsa y P es falsa; ¿Qué puede afirmarse de R y de Q?.
- Si Q ➔ Q ^ P es verdadera y P es falsa; ¿Qué puede afirmarse de Q?.
- Si R ^ P ➔ Q ^ P es falsa; ¿Qué puede afirmarse de P, Q y R?.
- Si (Q v R) → (P ^ Q) v R es falsa; ¿Qué puede afirmarse de P, Q y R?.
- Si (P ➔ Q) ➔ (R v P ➔ R v Q) es verdadera; ¿Qué puede afirmarse de P, Q y R?

10.12. Sean P, Q y R fórmulas. Determinar cuáles de las siguientes proposiciones son tautologías:

P ^ Q → P ^ R	(P → Q) → (¬Q → P)
P → P ^ Q	(P ←→ Q) ^ (P ^ ¬Q)
P ^ ¬(Q v P)	P ^ ¬((P v Q) v R)
(P → (Q v ¬P)) → ¬Q	P v (¬P v R)

Para la mayoría de los siguientes ejercicios propuestos se pueden realizar utilizando las variadas instrucciones de ciclos repetitivos (**goto**, **while**, **do-while** y **for**). Además, otro tanto podría también ser resueltos a través de arreglos.

10.13. Leer N números y determinar el promedio de sólo los números negativos.

10.14. Realizar la multiplicación de dos números leídos por el usuario, pero a través de sumas sucesivas.

10.15. Leer un total de 5 notas de un alumno cualquiera y escribir cuáles son las 2 peores.

10.16. Leer 2 números enteros positivos y determinar si uno es divisor del otro.

10.17. Escribir un programa que determine si un año es bisiesto. Un año es bisiesto si es múltiplo de 4 (por ejemplo 1984). Los años múltiplos de 100 no son bisiestos, salvo si ellos son también múltiplos de 400 (2000 es bisiesto, pero 1800 no lo es).

10.18. Dada la duración, en minutos, de una llamada telefónica, calcular el costo considerando que si es menor o igual a 3 minutos costará 1200,00, y si es por encima de este tiempo será de 1200,00 más el 10% de cada minuto adicional.

10.19. Hacer un programa que lea 3 números y nos indique si estos se encuentran en orden creciente o decreciente.

10.20. Hacer un programa que muestre la tabla de dividir de un número leído por la consola. Mostrarlos en forma decreciente.

10.21. Realizar un programa que permita leer 10 números primos cualesquiera y de cómo resultado el promedio de ellos.

10.22. Hacer un programa que permita calcular las 5 primeras parejas de números primos gemelos (dos números son primos gemelos si, además de ser primos, la diferencia entre ellos es exactamente igual a 2).

10.23. Diseñar un programa que permita calcular los 5 primeros números perfectos (un numero es perfecto cuando la suma de sus divisores, sin incluirlo, es exactamente el mismo número. Por ejemplo, el 6 es un número perfecto, porque sus divisores son 1, 2 y 3).

10.24. Realice un programa que lea por consola 5 números y que muestre como resultado, por consola, los dos menores.

10.25. Ingresar el día, mes y año en el formato numérico (dd-mm-yyyy → 02-11-2005), para luego mostrarlo en formato texto (2 de Noviembre del 2005).

10.26. Realizar un programa que permita ingresar un número determinado de segundos y lo convierta en horas, minutos y segundos.

10.27. Realizar un programa que permita ingresar la hora, minutos y segundos y que indique cuántos segundos en total son.

10.28. Diseñar un programa que permita ingresar la hora, minutos y segundos y que calcule la hora exacta en el siguiente segundo ("0=< H =<23", "0=< M =<59", "0=< S=<59").

10.29. Ingresar el número de un mes del año y determinar cuántos días tiene. Para el caso de Febrero, el programa deberá indicar que no cuenta con la información necesaria para dar la respuesta.

Ing. Fausto Lucchi

10.30. Realice un programa que lea por la consola un valor decimal (el monto de una compra) y que se calcule un descuento, considerando que si la compra es mayor a 1000 será de un 10%, y si es mayor a 5000 el descuento será de un 20%. Mostrar por consola de cuanto será el descuento y el monto total de la compra.

10.31. Calcular las raíces de una ecuación cuadrática, de la forma $ax^2+bx+c=0$ (deberá tomar en cuenta cuando la solución entre en el rango de lo imaginario).

10.32. Dada una figura cuadrilátera cualquiera, leer los valores de los cuatro lados (iniciando desde el lado izquierdo y siguiendo el sentido contra reloj), e indicar si dicha figura forma un cuadrado, un rectángulo o simplemente es una figura inválida.

10.33. Dado los valores de los dos catetos de un triángulo, calcular la hipotenusa y mostrarlo por pantalla.

10.34. Realizar un programa que lea un número natural y lo convierta en números romanos.

10.35. Leer 15 números enteros positivos, y que dé como resultado el máximo, el mínimo, la suma y el promedio de todos ellos.

10.36. Leer 6 notas de exámenes prácticos y 3 notas de exámenes parciales. Luego, indique cuál es el promedio de los prácticos y el promedio de los parciales. Además, mostrar un mensaje si el estudiante tiene derecho de asistir al examen final, de reparación o a ninguno de estos. NOTA: el estudiante tendrá derecho al examen final si el 20% del promedio de los prácticos más el 50% del promedio de los parciales da un valor mayor o igual a 3,5. Así mismo, tendrá derecho de asistir al examen de reparación si dicha suma anterior da como resultado un valor mayor o igual que 2,0.

10.37. Igual al anterior, pero se deberá indicar cuántas palabras comienzan con una vocal y terminan con esa misma letra. Ejemplo, la palabra "este" empieza con la letra 'e' y termina con la misma vocal.

10.38. Realice un programa que lea por consola una oración y una palabra cualesquiera. El programa deberá dar como resultado cuantas veces aparece la palabra ingresada en la oración. Ejemplo:
Oración: Esta casa es solo mía y es de mamá
Palabra: es
Resultado: Aparece 2 veces

10.39. Realice un programa que lea por la consola una oración y tres palabras. Luego deberá buscar, dentro de la oración, la primera palabra leída. Si se consigue, se deberá sustituir por la segunda palabra ingresada. En caso de que aparezca una segunda o más veces, se deberán sustituir por la tercera palabra ingresada por el usuario. Ejemplo:
Oración: esta materia esta cada vez más difícil, pero esta vez...
Palabra 1: esta
Palabra 2: la
Palabra 3: es
Resultado: la materia es cada vez más difícil, pero es vez...

10.40. Realice un programa que lea por consola una cadena de tipo "123+567" y efectuar la operación que indique la misma. En caso de que el usuario ingrese una cadena no válida, deberá mostrarse un mensaje de error indicándolo. Una cadena es válida si contiene dos operandos y en medio de estos un operador de los siguientes: suma (+), resta (-), multiplicación (*) o división (/). El programa deberá salirse solo cuando el usuario ingrese una cadena vacía.

10.41. Realice un programa que lea por la consola una oración o párrafo, e indicar, por consola, cuantas palabras posee dicha oración/párrafo. También deberá mostrar cuantas letras "a" y "s" (agrupar las mayúsculas y minúsculas). El programa terminará sólo cuando el usuario haya ingresado una oración vacía o párrafo vacío.

10.42. Realice un programa que lea por consola dos cadenas cualesquiera, y que de cómo resultado (imprimirla por la consola) una nueva cadena que alterne cada una de las palabras de ambas cadenas. Ejemplo:
Ingrese cadena 1: Hola cómo
Ingrese cadena 2: Mundo estás?
Resultado: Hola Mundo cómo estás?

10.43. Realice un programa que lea por la consola un párrafo y que de cómo resultado otro párrafo, con las mismas palabras, pero invertidas. O sea, la primera palabra será la última, la segunda palabra será la penúltima y así sucesivamente. Ejemplo:
Ingrese párrafo: esta casa es solamente mía
Resultado: mía solamente es casa esta

Ing. Faustoloccchi

10.44. Realizar un programa que lea una frase y que de cómo resultado una nueva frase pero con cada palabra invertida. Ejemplo:
hola mundo → aloh odnum

10.45. Realice un programa que lea una oración y una palabra. El programa generará una nueva oración borrando la palabra leída todas las veces que aparezca.

10.46. Realizar la siguiente modificación al problema anterior: indicar la posición de la palabra en la cadena (la palabra como tal y no como raíz).

10.47. Calcular la longitud de una cadena dada por el usuario (NO USAR la función "Length").

10.48. Hacer un programa que consistirá en situar ocho reinas en un tablero de ajedrez, de forma tal que ninguna de las reinas pueda actuar sobre cualquiera de las otras (que no se encuentren vertical, horizontal ni diagonalmente).

10.49. Se desea realizar un programa que realice las siguientes tareas: a) Leer una lista de números enteros b) Visualizar dichos números c) Preguntar al usuario si se desea ordenar la lista de forma creciente o decreciente.

10.50. Se tiene un arreglo A en donde se encuentran 10 números ordenados en forma ascendente, y se pide que el programa permita el ingreso de un número y lo coloque en la posición correcta dentro de dicho arreglo.

10.51. Diseñar un programa que calcule y almacene en un arreglo los 15 primeros números pares anteriores a un número entero dado por el usuario N, en donde N deberá ser mayor a 51 y menor que 100.

10.52. Hacer un programa que lea 10 apellidos y que los imprima en orden inverso al orden en cómo fueron ingresados.

10.53. Elaborar un programa que lea 100 números enteros positivos y que de cómo resultado el promedio de sólo los que sean múltiplos de 3.

10.54. Leer los datos de 15 alumnos (Nombre y nota final), ordenarlos por el mayor puntaje e imprimir dichos datos por la consola.

10.55. Realizar un programa que lea los datos de 15 trabajadores (Nombre, Horas laboradas y el Costo de cada hora). Luego, calcule por cada uno de ellos lo siguiente: a) Sueldo bruto b) Sueldo neto (sueldo bruto menos 5% de deducciones) c) Imprimir el sueldo bruto y el neto.

10.56. Hacer un programa que lea 30 números, los vaya agregando en un array, y que luego muestre el cuadrado de los números registrados en las posiciones pares.

10.57. Hacer un programa que registre 50 números en un array y que muestre los números registrados en las posiciones impares de forma decreciente sin tomar en cuenta el intervalo entre 25 y 30.

10.58. Leer un arreglo de tamaño N y determinar el mayor de los números, además indicar las veces que se repite el primero de los elementos.

10.59. Leer un arreglo de N elementos y luego introducir un número cualquiera. Determinar si el número leído se encuentra en el arreglo. En caso de ser así, indicar la(s) posición(es) donde se encuentra(n).

10.60. Dado un arreglo de N elementos, hallar la media, la moda y su frecuencia.

10.61. Dado un arreglo de N elementos, determinar si es palíndrome. Es decir, el dato contenido tiene el mismo significado leyéndolo en ambos sentidos.

10.62. Hacer un programa que genere una matriz identidad de orden NxN, en donde N es leído por el usuario.

10.63. Dada una matriz cualquiera, generar su respectiva matriz transpuesta.

10.64. Leer dos cadenas y determinar si son semejantes. Ejemplo: "ROMA" es semejante a la cadena "AMOR".

10.65. Dada una lista de 100 personas (Nombre, Apellido, Edad), se pide una relación de las personas mayores de 35 años.

10.66. Hacer un programa que permita el ingreso de 20 registros ($Nombre_x$, $Cedula_x$, $Seccion_x$). Luego, dada una sección cualquiera ingresada por el usuario, mostrar los nombres de los alumnos inscritos en la misma.

10.67. Hacer un programa que permita el ingreso de 20 registros ($Nombre_X$, $Telefono_X$, $Cedula_X$). Luego, dado un número de cédula cualquiera, mostrar teléfono y nombre de la misma.

10.68. Dado dos arreglos A y B (ambos unidimensionales) ordenados, generar un tercer arreglo C, producto de la intercalación ordenada de los elementos de los otros dos arreglos.
Ejemplo:

Ing. Fausto Cecchi

A = {3, 7, 13, 19}
B = {8, 9, 10, 12}
C = {3, 7, 8, 9, 10, 12, 13, 19}

10.69. Dado un arreglo de 10 elementos (leído por el usuario), leer un elemento y buscar la posición donde se encuentra.

10.70. Modificar el problema anterior, buscando no sólo la posición de la primera aparición del elemento, sino mostrar los subíndices (posiciones) en donde se encuentra el elemento, en caso de que este se repita.

Ing. Fausto Iocchi