Programación Orientada a Objetos con C#

Parte I Introducción a la Programación

Roger Pérez Chávez
Antonio Fernández Orquín
Airel Pérez Suárez
Raudel Hernández León

Universidad de Matanzas "Camilo Cienfuegos"
Universidad de las Ciencias Informáticas
Septiembre 2003

ISBN: 959-16-0231-6

Prólogo

El presente texto (o proyecto de libro en este momento) forma parte de un proyecto de investigación que se desarrolla entre dos universidades cubanas, la Universidad de Matanzas "Camilo Cienfuegos" (UMCC) a la que pertenecen los dos primeros autores y la Universidad de las Ciencias Informáticas (UCI) donde laboran como profesores Airel y Raudel.

Indirectamente participan otras dos universidades, la Universidad de la Habana (UH), Cuba, donde han sido formados tres de los cuatro autores y desde donde hemos recibido significativas influencias en cuanto a enseñanza de la programación, especialmente en la persona del profesor Dr. Miguel Katrib Mora. También es de destacar la participación de la Universidad de Murcia, España, a través del profesor Dr. Jesús J. García Molina, quien ha tenido la paciencia de revisar versiones anteriores realizándonos importantes y constructivos señalamientos.

Nota: No hacemos responsables a los dos profesores citados con anterioridad de ninguno de los errores que puedan existir en el texto.

¿Quién debería leer este texto?

De momento el presente texto tiene fines didácticos y principalmente se ha pensado para estudiantes y profesores de las especialidades de Informática y Ciencia de la Computación.

Si este es su primer enfrentamiento al apasionante mundo de la programación, no se preocupe, precisamente hemos pensado mucho en usted a la hora de elaborar el presente material. En otro caso, coincidimos con Archer en [13] cuando advierte que si un estudiante ha tenido alguna experiencia previa de programación pero no con el paradigma OO, debe estar prevenido: "¡la experiencia anterior con otros lenguajes no OO no le servirá de mucho!". La POO es una manera diferente de pensar en cómo diseñar soluciones a los problemas a través de una computadora. Finalmente si usted ha tenido algún contacto previo con la POO esperamos que el presente libro contribuya a madurar sus conceptos y lo motive para continuar profundizando en el arte de la programación y particularmente en la POO.

Si es usted profesor y ha pensado utilizar el texto en sus clases podrá apreciar que hemos querido desarrollar todos los capítulos de forma homogénea donde a través de situaciones de análisis se presentan los conceptos (preparación de las conferencias), se desarrollan casos de estudio para consolidar los conceptos presentados previamente (autopreparación de los estudiantes), luego se presentan ejercicios para desarrollar las clases prácticas y finalmente se sugiere bibliografía complementaria para profundizar en los tópicos presentados.

Motivación

La inspiración para escribir este texto surge al enfrentar la necesidad de impartir un curso para el cual no hemos podido encontrar un texto apropiado. En nuestro caso estamos hablando de un curso de Introducción a la Programación (IP) desde el enfoque de la Programación Orientada a Objetos (POO) y particularmente con el lenguaje de programación C#. Concretamente, nuestro objetivo más general con este texto es apoyar el desarrollo de una primera asignatura de Programación en nuestras Universidades con el enfoque del paradigma Orientado a Objetos (OO) (principalmente en carreras de Ingeniería Informática y Ciencia de la Computación).

El presente texto se encamina a transmitir conceptos básicos de la POO como objeto, clase, relaciones de uso y asociación entre clases y objetos y diseño dirigido por responsabilidades de una manera independiente al lenguaje pero de forma práctica resolviendo problemas a través de aplicaciones simples en modo consola con C#. Paralelamente a estos conceptos se abordan los elementos básicos de programación correspondientes a un curso clásico de IP como secuencias lineales y análisis de casos (estructuras de control alternativas), estructuras de control repetitivas, arreglos, iteración, recursividad y muy brevemente elementos muy básicos de búsqueda y ordenamiento.

En ningún momento el presente libro pretende hacer aportaciones a la POO, ni a los conceptos básicos de un curso tradicional de IP pues en ambas casos existen muy buenos textos que en muchos casos han sido fuente de inspiración [1, 3, 8, 9] y consulta para la realización del presente libro. En términos de consulta hemos realizado una revisión exhaustiva de la bibliografía existente para desarrollar los conceptos que abordamos en este texto y para seleccionar los correspondientes ejercicios de cada sección. Dejemos claro que

tampoco es nuestro objetivo que el presente texto constituya una guía de referencias del lenguaje seleccionado (C#) pues en este sentido también existen disímiles trabajos con muy buena calidad que también hemos consultado frecuentemente para la realización del presente libro [10, 12, 13].

Insistimos en que nuestro objetivo es intentar encontrar el momento y la forma precisa de presentar cada concepto y motivar este proceso a través de la selección y formulación adecuada de los correspondientes ejercicios.

Antecedentes

En el caso de los dos primeros autores, escucharon hablar por primera vez de comenzar la enseñanza de la programación desde cero con objetos en el evento COMAT de la UMCC en Noviembre de 1995, precisamente a través del Dr. Miguel Katrib. A partir de este momento ha sido su preocupación constante cómo llevar a cabo este reto y han experimentados diferentes propuestas metodológicas, tanto en la UMCC como en la Universidad Católica de Brasilia, Brasil (UCB).

Sin embargo, el catalizador para desarrollar el presente trabajo fue el curso de *.NET y Web Services* en el que participó el primero de los autores, una extraordinaria iniciativa del profesor Katrib y el grupo de investigación WebOO de la UH. A partir de este curso, donde se tuvo el primero encuentro con C#, lenguaje que nos ha convencido para comenzar desde cero OO, se desarrolló una experiencia de un curso de OO en la UMCC en la que se sustenta el desarrollo de la presente obra.

En el caso de los profesores Airel y Raudel, tuvieron la experiencia personal de recibir en sus estudios de pregrado, un primer curso de programación con en el enfoque OO y *Delphi* como lenguaje, también a través del profesor Katrib.

Combinando las experiencias de todos los autores es que hemos constituido un equipo de trabajo que ha laborado intensamente durante tres meses (aproximadamente) en la obtención de este resultado parcial que hoy exponemos.

Organización de libro

Coincidimos con Budd [1] cuando afirma que "la mayoría de los libros de introducción a la POO cometen el error de comenzar con la sintaxis de un lenguaje en particular antes de presentar la "filosofía" OO. De hecho, a menudo resulta sorpresivo para no pocas persona que haya un enfoque de pensamiento OO acerca de la computación y la resolución de problemas y que además éste sea bastante independiente de la sintaxis usada para aplicar la técnica".

Inclusive, confirmando esta afirmación vemos como los estudiantes generalmente se refieren al curso de POO de la siguiente forma: "estoy dando un curso de C++" (o Delphi, Java, C#, etc.). Es por esta razón que nos inspiramos en el libro de Budd [1] para definir un primer capítulo dedicado a establecer una base para fomentar una manera de pensar OO. Este primer capítulo forma parte de un primer tema que hemos denominado: "Secuenciación. Aplicaciones simples en modo consola". En este primer tema se desarrollan tres capítulos en los que se abordan los elementos principales del modelo OO, luego se realiza la implementación de las clases en C# y posteriormente se desarrollan aplicaciones simples en modo consola donde apenas aparezcan secuencias lineales de instrucciones.

El libro lo constituyen en total cinco temas, el segundo se denomina: "Análisis de casos. Encapsulamiento" y en el mismo se comienzan presentando situaciones que necesitan del análisis de casos para presentar las estructuras de control alternativas, luego se definen clases que necesiten definir componentes que a su vez son objetos y finalmente se aborda el concepto de Encapsulamiento y muy básicamente los conceptos de Programación por contratos y Excepciones para garantizar la integridad de los objetos.

El tercer tema se denomina: "Iteración. Arreglos" y en este se aborda la noción de secuencia y su representación a través de arreglos unidimensionales y sobre éstos se implementan algoritmos básicos que precisan de las estructuras de control repetitivas. También se presentan diversos esquemas de recorridos y búsquedas y por último los arreglos bidimensionales y las variantes de tablas regulares e irregulares.

El cuarto tema se denomina: "Recursividad. Búsqueda y Ordenamiento" y en el mismo se presenta el esquema general de una iteración, las sucesiones y el concepto de recursividad para finalizar con elementos muy básicos de ordenamiento y búsqueda.

Finalmente, el capítulo V se destina a presentar elementos complementarios del lenguaje y el entorno de trabajo. De momento apenas se presentan los conceptos básicos de la plataforma de desarrollo Microsoft .NET, fundamento del lenguaje C#.

Es de destacar que hemos intentado realizar una selección minuciosa de los ejercicios reformulándolos en muchos casos en función de orientar a los estudiantes hacia una solución OO.

Finalmente coincidimos con García Molina en [9] en nuestro objetivo de "educar a los alumnos en el rigor, no creando programadores compulsivos, ansiosos por sentarse ante la máquina para escribir directamente el programa y establecer con el ordenador una batalla estúpida hasta lograr el supuesto programa que proporciona la solución al problema planteado. Queremos que los alumnos estén preocupados por obtener programas correctos y con buenas propiedades internas (legibilidad, estructuración, etc.). Una vez que el diseño ideado merece suficiente confianza, se traduce y se ejecuta en la computadora.

Estado actual y trabajo futuro

La premura por la necesidad de tener una primera impresión para utilizar el texto y el enfoque subyacente en el presente semestre en dos grupos de la UMCC y cuatro de la UCI nos han llevado a obtener este resultado parcial que refinaremos en el transcurso del semestre y para lo cual reclamamos la importante e imprescindible ayuda de los lectores. Sin dudas, sus sugerencias serán muy valiosas para nosotros.

De antemano, en casi todos los temas estamos conscientes que nos faltaron elementos por abordar. Por ejemplo:

- En el tema II aún hay que profundizar en la parametrización de los métodos, tipos de parámetros, etc. Así como las primeras nociones del análisis descendente en la solución de problemas y las enumeraciones como valioso recurso pata la construcción de tipos de dato.
- En el tema III es posible profundizar sobre el análisis descendente, la implementación de las relaciones de asociación y agregación, así como la determinación de secuencias intermedias.
- En el tema V es posible incorporar elementos complementarios sobre como documentar nuestros códigos y por ende presentar elementos de XML y atributos.

A su disposición:

Roger Pérez Chávez (UMCC) (roger.perez@umcc.cu)

Antonio Fernández Orquin (UMCC) (antonio.fernandez@umcc.cu)

Airel Pérez Suárez (UCI) (airel@uci.cu)

Raudel Hernández León (UCI) (raudel@uci.cu)

18 de Septiembre 2003

Tabla de contenidos

Tema I

I. SECUENCIACION. APLICACIONES SIMPLES EN MODO CONSOLA

Objetivos

- **Identificar** objetos y clases en enunciados de problemas de poca complejidad.
- **Diseñar** diagramas de clases de poca complejidad.
- **Desarrollar** algoritmos de poca complejidad que utilicen la secuenciación como técnica de descomposición.
- **Implementar** clases en C# que apenas se relacionen con otras clases a través de parámetros de los métodos.
- **Obtener** aplicaciones simples en modo consola a través de un Ambiente Integrado de Desarrollo (IDE-*Integrated Development Environment*).

Contenido

I.1 Elementos principales del modelo Orientado a Objetos (OO).
I.2 Implementación de clases en C#. Secuenciación.
I.3 Aplicaciones simples en modo consola. Secuenciación.

Descripción

El objetivo integrador de este tema es obtener aplicaciones simples en modo consola, a partir del enunciado de un problema de poca complejidad.

El tema comienza con la presentación de una situación de análisis para caracterizar los elementos principales del modelo OO. Posteriormente se orientan y desarrollan ejercicios prácticos para aplicar los conceptos principales presentados en esta primera parte del tema.

En un segundo momento pasamos a la implementación de las clases y para ello se presentan los recursos en un lenguaje orientado a objetos.

Posteriormente se muestran los elementos esenciales de una aplicación consola construida en un ambiente integrado de desarrollo (IDE Integrated Development Environment) y se construyen aplicaciones simples. Concluye esta parte entonces con la orientación y desarrollo de ejercicios integradores de los contenidos del tema.

Capítulo I.1

I.1 Introducción al modelo Orientado a Objetos

"Casi 20 años de desarrollo de software (incluyendo los últimos ocho con lenguajes Orientados a Objetos (OO)) me han enseñado que los conceptos de la Programación Orientada a Objetos (POO), cuando se aplican correctamente, realmente cumplen con su promesa" [13].

En este capítulo se presentan los elementos básicos del modelo OO. Para ello en la primera sección se enuncia y analiza una situación de nuestro acontecer y a partir de la misma, en el resto del capítulo se caracterizan entre otros los conceptos de objeto, clase, responsabilidades y relaciones entre clases.

Un libro centrado en el lenguaje no es una buena práctica para un primer contacto con la POO ni con la programación en general. Los libros que centran su atención en el lenguaje son apropiados para personas que ya cuentan con experiencia en POO, pues solamente tendrían que asimilar los detalles técnicos del nuevo lenguaje.

Estas son las razones que justifican tener un primer contacto con los conceptos básicos del modelo OO antes de interactuar con el lenguaje. Incluso en todo el desarrollo del libro se hace énfasis en algunos aspectos de la modelación como paso previo a la programación.

Antes de presentar la situación de análisis como parte del desarrollo de la primera sección es interesante reflexionar sobre las siguientes ideas: "… la POO se parece mucho a la cerveza…. A la mayoría de la gente que la prueba por primera vez no le gusta, y podría cuestionar la salud mental de quienes cuentan sus alabanzas. Que te hice …---dirían--- para que me hicieras beber esto? Algún tiempo después, sin embargo, se cultiva un gusto por la cerveza en quienes continúan bebiéndola. La POO como la cerveza, es un gusto que se adquiere con el tiempo. Otro parecido de la POO con la cerveza: puede provocar desorientación, causar náuseas y una visión alterada del mundo" [2].

Efectivamente, la experiencia confirma que en un primer enfrentamiento al modelo OO se producen varios tipos de reacciones, a algunos les parece natural y se sienten muy motivados por el acercamiento de este modelo al mundo real mientras que a otros les atormenta el volumen de conceptos que se relaciona y llegan a sentirse verdaderamente perdidos. Durante el transcurso de la comprensión del modelo OO, como todo proceso de adquisición de conocimientos y habilidades donde existe una curva de aprendizaje, también hay quienes consiguen avanzar más rápido que otros pero finalmente donde existe absoluta coincidencia es en el apasionamiento de aquellos que consiguen interiorizar y comprender las bondades del referido modelo.

"La POO es mucho más que una frase comercial (aunque haya llegado a ser así por culpa de algunas personas), una nueva sintaxis o una interfaz para la programación de aplicaciones (API Application Programming Interface). La POO es un conjunto de conceptos e ideas. Es una manera de pensar en el problema al que va dirigido nuestra aplicación y de afrontarlo de modo más intuitivo e incluso más productivo. El enfoque OO es más intuitivo y más cercano a cómo muchos de nosotros pensaríamos en la manera de afrontar un problema" [13].

"Es probable que la popularidad de la POO derive en parte de la esperanza de que esta nueva técnica, cómo pasó con muchas innovaciones previas en el desarrollo del software de computadoras, sea la clave para incrementar la productividad, mejorar la confiabilidad, disminuir los vacíos, reducir la deuda nacional. Aunque es verdad que se obtienen muchos beneficios al usar técnicas de POO, también es cierto que programar una computadora es todavía una de las tareas más difíciles jamás emprendidas por la humanidad; llegar a ser un experto en programación requiere talento, creatividad inteligencia, lógica, habilidad para construir y usar abstracciones y experiencia, aún cuando se disponga de las mejores herramientas" [1].

En el caso particular de la POO es imprescindible el desarrollo de habilidades lógicas del pensamiento humano como identificar, clasificar y generalizar pues uno de los objetivos más importantes de este paradigma es la identificación de clases como generalizaciones de dominios de objetos o la identificación de objetos que luego se agruparían en clases a través de procesos de clasificación y generalización. Otra habilidad importante que requiere el paradigma OO es en cuanto a determinar y establecer colaboraciones

entre objetos y/o clases. La POO es un una nueva forma de pensar acerca de lo que significa programar una computadora y acerca de cómo se puede estructurar la información dentro de la misma.

"¿Es la POO un mejor paradigma que otros? En cierto sentido sí lo es. ¿Es una cura de todos nuestros problemas? No, no lo es. ¿Significa que la denominada "crisis del software" desaparecerá? Probablemente no. Pero entonces, ¿qué es lo grande de la POO?" [3].

En el modelo OO, "en lugar de tratar de modelar un problema en algo familiar a la computadora se trata ahora de acercar la computadora al problema. Es decir, modelar la realidad del problema a través de entidades independientes pero que interactúan entre sí y cuyas fronteras no estén determinadas por su instrumentación computacional sino por la naturaleza del problema. Estas entidades serán denominadas objetos por analogía con el concepto de objeto en el mundo real" [3].

"Esto es tal vez lo bello de la POO, la sencillez de sus conceptos. Resolver problemas consiste en definir objetos y sus acciones y entonces invocar las acciones enviando mensajes a los objetos que ocultan las características internas de cómo llevan a cabo estas acciones. Esta forma de resolver problemas luce familiar y lógica y ha sido utilizada por otras disciplinas científicas ya que en nuestra experiencia diaria nosotros "manipulamos objetos"" [3].

I.1.1 Identificación de Objetos y Clases

Para mostrar algunas de las principales ideas del modelo de objetos, se presenta a continuación una situación de la vida diaria.

Situación de análisis

Un pequeño niño de aproximadamente dos años de edad puede decirle a su papá: esta es mi mamita, estos son mis juguetes y señalar hacia un velocípedo, dos camiones y tres soldaditos, esto es una silla, esto es una mesa, esto es un jabón, este es mi cepillo, etc. Incluso, muchas veces cuando se encuentra en el carro de su papá (VW Golf año 85), es capaz de decirle al mismo que arranque, que pite o que ponga la música.

De igual forma, al interrogársele, el niño dice su nombre (Alejandro) y el de sus padres (Gloria y Alexis). También si a este niño se le indica que vaya a lavarse los dientes sin dudarlo seleccionará su propio cepillo (aún cuando no se pueda cepillar los dientes correctamente). Este mismo niño es capaz de encender y apagar un ventilador o un televisor e incluso en alguna ocasión apagar el carro de su papá. Aproximadamente con tres años de edad es capaz de decir entonces: "el carro de papá es rojo", o que "el carro de tío es azul", etc.

Por lo general, en su casa y en un día normal, llegada la hora de dormir, el referido niño le dice a su mamá o a su papá: tengo sueño; entonces el receptor de este mensaje (mamá o papá) se ocupa de dormir al niño (o solicita la ayuda del otro) y para ello debe llevar a cabo una serie de acciones (prepararle la leche, cambiarle la ropa, acostarlo con la cabecita sobre la almohada, darle a tomar el pomo de leche e incluso acostarse a su lado hasta que se duerma).

Pregunta: ¿Por qué un niño es capaz de decir o hacer las cosas que se relacionan en la situación de análisis?

Aún cuando no se tiene ningún conocimiento de psicología infantil, se puede asegurar que un niño es capaz de decir o hacer esas cosas porque desde edades muy tempranas los seres humanos son capaces de *identificar* y *clasificar* elementos o entidades (*objetos*) del mundo que le rodea. Incluso, de determinar las características y las funcionalidades (*responsabilidades*) de determinados elementos; así como relacionarse con el entorno y particularmente comunicarse con sus semejantes.

Al igual que en el quehacer diario, en el modelo OO cobran vital importancia procesos lógicos del pensamiento como *identificación* y *clasificación*.

Ejemplo: Determinar algunos de los objetos que aparecen en la situación de análisis presentada con anterioridad y las responsabilidades que aparezcan de forma explícita. Establezca además algún tipo de clasificación para los objetos hallados.

La solución del ejemplo se presenta en la siguiente tabla:

Clasificación	Objetos	Responsabilidades
`Niño`	`alejandro`	Nombre Edad Juguetes Cepillarse los dientes Encender y apagar algunos efectos electrodomésticos
`Mamá`	`gloria`	Nombre Dormir al niño
`Papá`	`alexis`	Nombre Arrancar el carro Tocar el pito Dormir al niño
`Carro`	`VW`	Marca Modelo Año de fabricación Color Arranca Pita

Vale aclarar que en la situación planteada existen algunos objetos que no se han identificado, e incluso de los identificados no se señalaron todas sus responsabilidades con el objetivo de no sobrecargar mucho el texto.

De forma inconsciente, al extraer los objetos `alejandro, gloria, alexis y VW` se han clasificado en `Niño, Mamá, Papá y Carro` respectivamente y se han determinado características y funcionalidades (responsabilidades) que corresponden a ellos.

De forma general todos los niños del barrio son ejemplares de la clasificación Niño, todos los padres de las clasificaciones Papá o Mamá y los carros ejemplares de Carro. En el modelo OO a estas clasificaciones se les denomina clases. De cada clase se pueden tener varios ejemplares los cuales se denominan objetos. A continuación se presentan formalmente los conceptos de clase y objeto.

I.1.2 Clases y Objetos

Los conceptos básicos del paradigma OO son clase y objeto. En el modelo OO se combina la estructura y el comportamiento de los datos en una única entidad, los objetos. Los objetos son simplemente entidades que tienen sentido en el contexto de una aplicación (dominio del problema).

Todos los objetos son ejemplares o instancias de alguna clase, los términos instancia y objeto son intercambiables. Los objetos de una misma clase tienen las mismas responsabilidades. Los objetos con las mismas responsabilidades pueden ser agrupados en una clase. Las clases son abstracciones que generalizan dominios de objetos.

Un proceso básico que enfrentamos en la resolución de problemas es la abstracción, o sea la eliminación de los detalles irrelevantes, centrándonos únicamente en los aspectos esenciales del problema [9] (dominio del problema).

La POO se basa en el modelo OO. Los objetos del mundo real son representados como objetos en el diseño y en la programación. En el modelo OO, toda entidad del dominio del problema se expresa a través del concepto de objeto y éstos se generalizan a través del concepto de clase. Entendemos por dominio del problema, al problema que se intenta resolver en término de complejidades específicas, terminologías, retos, etc. [13]

Ejemplo: Determinar los objetos presentes en un juego de ajedrez (idea original en [4]).

Comúnmente, como respuesta a este ejemplo se enuncia el peón, el caballo, la torre, etc., pero sin dudas estos son conceptos o abstracciones que agrupan a las piezas concretas que aparecen en el juego de ajedrez, es decir, peón, caballo y torre entre otras, constituyen clases, que entre sus responsabilidades tienen el color, la

posición que ocupan, la forma de avanzar y comer, etc. Los objetos son entonces los 16 peones (8 negros y 8 blancos), los 4 caballos, las 4 torres, el tablero, etc.

Durante una partida de ajedrez, dos peones negros que hayan sido comidos tienen exactamente las mismas características y sin embargo son objetos distintos. Precisamente, una de las características de los objetos que permite su identificación es su identidad.

"Identidad significa que los datos son divididos en entidades discretas y distintas, denominadas objetos" [4]. Además los objetos pueden ser concretos como el niño alejandro de la sección anterior o conceptuales, como el método de Gauss para determinar raíces en sistemas de ecuaciones lineales.

Todo objeto tiene su propia identidad, que le es inherente, significa que los objetos se distinguen por su propia existencia y no por las propiedades descriptivas que puedan tener. Es esto lo que lo diferencia de los otros objetos semejantes. En otras palabras, dos objetos son diferentes, incluso si todos los valores de sus características son idénticos.

Sin dudas que la forma de cada una de las piezas, el color, la manera de avanzar y comer, en conclusión, las características (forma y color) y funcionalidades (manera de avanzar y comer) de cada una de las piezas, que no son más que sus responsabilidades, facilitaron determinar los objetos en el ejemplo anterior.

I.1.3 Diseño dirigido por responsabilidades

El concepto de diseño dirigido por responsabilidades es importante en las etapas tempranas de la solución de problemas y cobra un alto valor notar que el comportamiento de los objetos se puede estructurar en términos de las responsabilidades de los mismos [1]. Es por ello que en la identificación de las clases y objetos presentes en la situación de análisis se ha considerado con fuerza la determinación de las mismas.

Los objetos analizados en la situación de análisis de la sección I.1 tienen un estado determinado en dependencia del valor de sus características. Por ejemplo, el niño tiene una edad determinada y al cumplir año esta aumenta en uno, de igual forma el carro del papá al pintarse puede cambiar de color. Al mismo tiempo, estos objetos tienen un comportamiento que viene dado por las acciones que pueden realizar. Por ejemplo, la mamá duerme al niño, el niño se cepilla los dientes, etc.

Podemos concluir que el estado de un objeto viene dado por el valor de sus características y el comportamiento del objeto por las acciones u operaciones que realiza. Es decir las responsabilidades podemos dividirlas en dos grupos, aquellas que determinan el estado (atributos) y las que determinan el comportamiento (métodos).

Otros autores denominan a los atributos de diversas formas: campos de datos o simplemente campos (fields), propiedades o variables; mientras que a los métodos también les suelen llamar: rutinas o funciones miembro.

Los atributos son valores almacenados por los objetos de una clase mientras que los métodos son secuencias de pasos para determinar un comportamiento. Los métodos pueden ser llamados (invocados) en principio, solamente desde dentro de la clase o a través de una instancia u objeto de la misma.

Como se verá en secciones posteriores, es útil distinguir los métodos que tienen efectos colaterales, es decir, que pueden cambiar el estado del objeto (métodos de transformación), de los que apenas consultan el valor de algún atributo o realizan un cálculo funcional sin modificar ningún el estado del objeto (método de consulta o acceso).

I.1.4 Relaciones entre clases

Situación de análisis

Analice los siguientes planteamientos:

La clase `Niño` tiene una responsabilidad que es "Cepillarse los dientes", la cual es imposible realizar sin el uso de un objeto o instancia de una clase que se podría denominar `Cepillo de dientes`.

`alejandro` (instancia de la clase `Niño`) es hijo de `alexis` (instancia de la clase `Papá`) y a su vez `alexis` es padre de `alejandro`.

Aunque no se ha expresado explícitamente, las clases `Niño`, `Mamá` y `Papá` tienen responsabilidades comunes por lo que pueden a su vez ser agrupadas en una clase más general `Persona`.

Pregunta: ¿Existen las clases y los objetos de forma aislada?

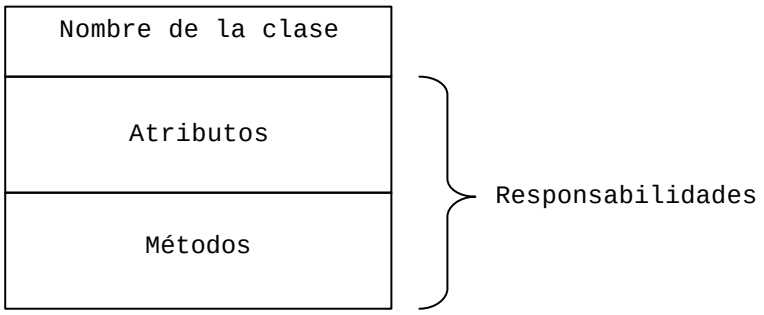
Claro está que no. En la situación anterior se muestra relaciones entre el niño y el cepillo de dientes, el niño y el padre, etc.

Es posible establecer relaciones entre dos clases por varias razones como se muestra más adelante. Una *relación* es una conexión entre elementos y entre las más importantes tenemos las *dependencias*, las *asociaciones* y las *generalizaciones*; gráficamente, una relación se puede representar por una línea, usándose diferentes tipos de línea para diferenciar los tipos de relaciones [26].

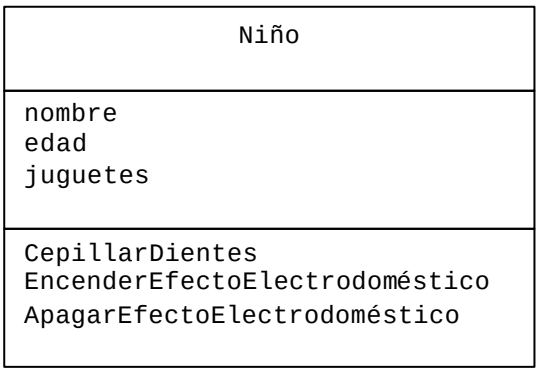
Por otra parte, hasta el momento se han presentado algunos elementos básicos del modelo OO. Específicamente se ha hecho hincapié en los conceptos de clase, objeto y responsabilidades de éstos. Los mismos se han descrito con ejemplos y prosa. Sin embargo, esta forma no permite abordar tópicos más complejos como las relaciones entre clases. Es por ello que se precisa de un formalismo para expresar modelos de objetos que sea coherente, preciso y de fácil formulación.

Sin dudas que las representaciones gráficas tienen gran importancia e influencia en la expresión de los conceptos. Por tanto, a partir de este momento se utilizarán los diagramas de clase, que constituyen una notación gráfica formal para la representación de las clases y sus relacionamientos. Estos a su vez son concisos, fáciles de entender y funcionan bien en la práctica.

En la siguiente figura se muestra el esquema general para la representación de una clase en la notación seleccionada. Nótese como esta notación deja clara la diferenciación entre los distintos tipos de responsabilidades dedicándole una sección a cada uno de ellos.



Seguidamente se presenta una variante de representación de la clase `Niño` en la notación seleccionada.



Note como en la representación de la clase Niño se han utilizado nombres simbólicos para denotar las responsabilidades. Por ejemplo, para la responsabilidad “Cepillarse los dientes”, el nombre simbólico (identificador) CepillarDientes. Por otra parte, los identificadores de atributos se han denotado comenzando en minúscula, mientras que los identificadores de los métodos comenzando con mayúscula. Todo esto de momento se verá como un simple convenio, sin embargo, en los siguientes capítulos resultará de gran utilidad.

I.1.4.1 Relación de dependencia o uso

Retomando el primer planteamiento de la situación de la sección I.1.4.

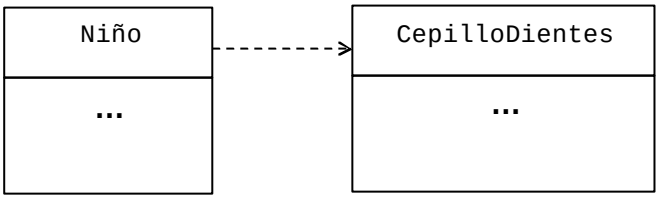
Situación de análisis

La clase `Niño` tiene una responsabilidad que se ha denotado `CepillarDientes`, la cual es imposible de realizar sin el uso de un objeto o instancia de una clase que podríamos denominar `CepilloDientes`.

Pregunta: ¿Qué tipo de relación se establece entre las clases `Niño` y `CepilloDientes`?

Nótese que esta relación se establece en una sola dirección, en este caso las instancias de la clase `Niño` se *auxilian* o *usan* una instancia de la clase `CepilloDientes` para llevar a cabo (*ejecutar*) el método `CepillarDientes`.

Este tipo de relación se denomina *dependencia* y establece una *relación de uso* que declara que un cambio en la especificación de un elemento (por ejemplo, la clase `CepillarDientes`) puede afectar a otro elemento que la utiliza (por ejemplo, la clase `Niño`), pero no necesariamente a la inversa [26]. Las dependencias se usarán cuando se quiera indicar que un elemento utiliza a otro. Gráficamente, una dependencia se representa con una línea discontinua dirigida hacia el elemento del cual se depende como se muestra a continuación.



Nota: En la figura se han sustituido las responsabilidades de las clases involucradas por "**...**" debido a que el interés se ha centrado en la relación entre las clases. A partir de este momento siempre que algún recurso no centre la atención se realizará la correspondiente sustitución por "**...**".

I.1.4.2 Asociación

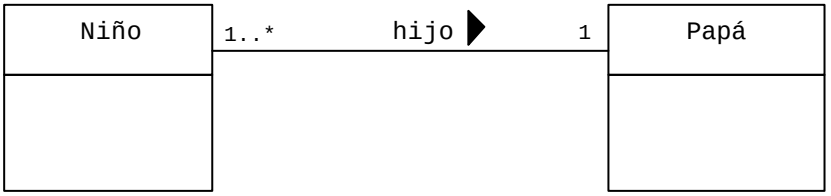
Situación de análisis

`Alejandro` (instancia de la clase `Niño`) es hijo de `Alexis` (instancia de la clase `Papá`) y a su vez `Alexis` es padre de `Alejandro`.

Pregunta: ¿Qué tipo de relación se establece entre las clases `Niño` y `Papá`?

Nótese que entre las clases `Niño` y `Papá` existe una relación bidireccional donde todo objeto de la clase `Niño` es hijo de un objeto de la clase `Papá` y viceversa, todo `Papá` es padre de al menos un `Niño`.

Este es un ejemplo de relación de *asociación* que se establece entre las clases `Niño` y `Papá`. Una asociación es una relación estructural que especifica que los objetos de una clase están conectados con los objetos de otra; dada una asociación entre dos clases, se puede navegar desde un objeto de una clase hasta un objeto de la otra y viceversa [26]. Gráficamente, una asociación se representa como una línea continua que conecta a las clases involucradas como se muestra seguidamente.



Las asociaciones interrelacionan clases y objetos y son intrínsecamente bidireccionales. A las asociaciones se les asigna un nombre (hijo) y este se lee en la dirección que indica el símbolo que le sucede (▶) o le precede (◀).

Nótese además en el diagrama anterior la existencia de "1..*" del lado de la clase Niño y de "1" del lado de la clase Papá. Esta es la forma de expresar que un Niño es hijo de un solo Papá ("1") y todo Papá tiene uno o varios hijos ("1..*"). Estos cuantificadores de la relación son los que determinan la cardinalidad o multiplicidad de la misma.

La cardinalidad o multiplicidad es un elemento muy importante que se debe tener en cuenta en las relaciones de asociación ya que ésta permite determinar cuantas instancias de un lado pueden estar presentes en una instancia del otro y viceversa.

I.1.4.3 Generalización

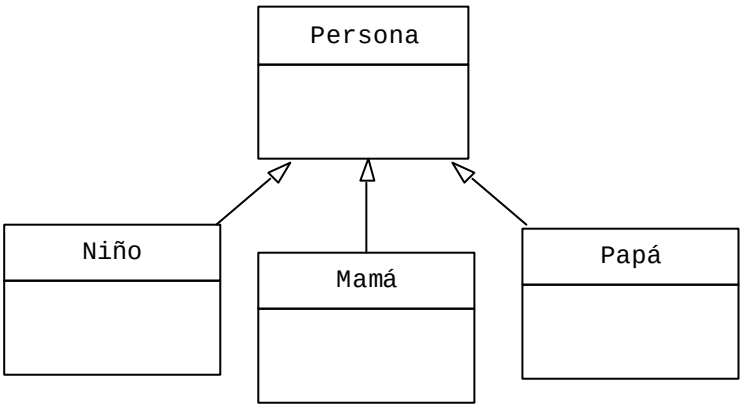
Situación de análisis

Aunque no se ha expresado explícitamente, las clases `Niño`, `Mamá` y `Papá` tienen responsabilidades comunes por lo que pueden a su vez ser agrupadas en una clase más general `Persona`.

Pregunta: ¿Qué tipo de relación se establece entre las clases `Niño`, `Mamá`, `Papá` y `Persona`?

Como en la clase `Persona` se agrupan un conjunto de responsabilidades comunes a `Niño`, `Mamá` y `Papá` es posible decir que entre estas tres clases y `Persona` se puede establecer una relación "es un". Por ejemplo, `Niño` "es una" `Persona`. Este tipo de relación se define como *generalización*.

Esta nueva relación ("es un") determina la existencia de superclases (`Persona`) que a su vez permiten establecer generalizaciones y de subclases (`Niño`, `Mamá` y `Papá`) que a la vez son nuevas clasificaciones (subdominios) dentro del dominio más general. Gráficamente, esta nueva relación se representa como una línea continua dirigida, con una punta de flecha vacía, apuntando a la superclase [26].



Sin dudas que los elementos asociados con las relaciones entre clases le imprimen cierto grado de complejidad al modelo OO y depende mucho de un buen nivel de abstracción y de la definición precisa del dominio del problema. No obstante, no se preocupe porque este no es el objetivo central del texto. Además, cada una de estas relaciones será abordada en detalle en futuros capítulos y secciones (profundizando en su implementación en un lenguaje en particular) e incluso, la relación de generalización será analizada con profundidad más allá de los límites del presente texto. Particularmente la relación de dependencia o uso será analizada a partir del siguiente capítulo (I.2), la asociación y la agregación se abordan partiendo del primer capítulo del tema III y por último la generalización en el tema VI (Parte II del presente texto).

Situación de análisis

Profundizar en la relación de generalización y su implementación, exceden los límites de este texto.

Pregunta: ¿Por qué enunciar el concepto de relación de generalización en este capítulo?

En la respuesta a esta pregunta existen dos razones: primero que el concepto de generalización es fundamental en el modelo OO, que es la base de esta propuesta y en segundo lugar porque se está intentando ver la programación como un sistema donde existe estrecha vinculación e interrelación entre cada una de las partes componentes. Además, la generalización es también una habilidad lógica del

pensamiento humano. Este concepto será abordado en un segundo texto, que tiene por objetivo general profundizar en el modelo OO.

Por otra parte, uno de los principios didácticos en que se fundamenta esta propuesta será la creación de expectativas en los lectores que se inician en el fascinante arte de la programación, claro que luego hay que trabajar para satisfacer en todo momento estas expectativas por lo que no se pretende de modo alguno dejarlos desamparados. Es un anhelo de los autores realizar un trabajo semejante para desarrollar el referido segundo texto. Aún cuando existen excelentes libros que podrían utilizarse, la mayoría de éstos no centra su objetivo en los lectores principiantes.

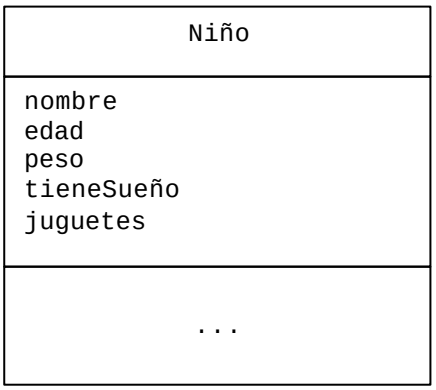
I.1.5 Clasificación de los atributos

Desde que fueron determinados los primeros objetos en la situación de análisis de la sección I.1.1 se hizo énfasis en clasificarlos (Por ejemplo, `Niño alejandro`).

Pregunta: ¿Es posible clasificar los atributos de forma general?

Claro que sí, de la misma forma que se han clasificado los objetos analizados anteriormente, se puede clasificar cualquier atributo de acuerdo a los valores que pueda almacenar y a las operaciones que se puedan realizar con estos valores, definiéndose de esta forma el concepto de *tipo de dato*.

Seguidamente se muestra una variante de la clase `Niño` donde se incluyen nuevos atributos que pudiera tener y posteriormente se determinará el tipo de dato de cada uno de ellos.



Para clasificar los atributos de `Niño` se presenta la siguiente tabla donde se relacionan posibles valores a almacenar por cada uno.

Atributos	Valores
`nombre`	`Alejandro` ó `Luis` ó `Ana` ó …
`edad`	`1` ó `2` ó `3` ó `4` ó …
`peso`	`16.5` ó `28` ó `35.5` ó …
`tieneSueño`	`si` ó `no`
`juguetes`	`velocípedo, camión, soldadito, pelota, …`

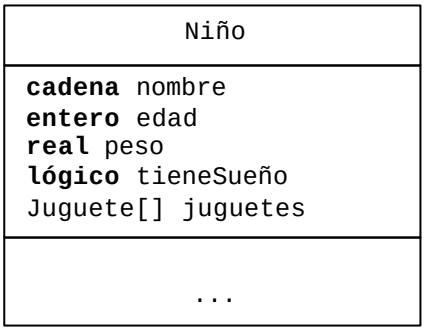
Como puede verse el atributo `nombre` toma como valores secuencias o combinaciones de letras (*cadenas de caracteres*), por el momento este tipo de valores se clasificará como tipo de dato **`cadena`**. Los atributos `edad` y `peso` toman valores numéricos, diferenciándose en que la `edad` toma valores del dominio de los números enteros (tipo de dato **`entero`**) y el `peso` toma valores reales (tipo de dato **`real`**).

El atributo `tieneSueño` toma dos posibles valores (`si` ó `no`) que se pueden hacer corresponder con dos posibles estados (`verdadero` o `falso`) definiendo el tipo de dato **`lógico`**.

Note que hasta el momento los atributos analizados pueden almacenar un único valor (a la vez) de su correspondiente tipo de dato. Sin embargo, el atributo `juguetes` puede almacenar varias instancias de la

clase `Juguete`, todas bajo un mismo nombre (`juguetes`). Es decir, el tipo de dato del atributo `juguetes` es un conjunto de instancias de la clase `Juguete` que se denotará por `Juguete[]` (en general `<TipoDeDato>[]`).

En la siguiente figura se refina la clase `Niño`, añadiéndose el tipo de dato de cada uno de sus atributos (*declaración de los atributos*).



Pregunta: ¿Por qué unos tipos de dato se encuentran resaltados en negrita (**`cadena`**, **`entero`**, **`real`** y **`lógico`**) y otros no (`Juguete`)?

Esto se debe a que existe a su vez una clasificación entre los tipos de dato en: *simples* (básicos, primarios o primitivos) (**`cadena`**, **`entero`**, **`real`** y **`lógico`**) y *clases* (`Juguete`).

Por el momento estos tipos de dato serán utilizados en los ejemplos de las próximas secciones.

I.1.6 Acceso a las responsabilidades de los objetos. Parametrización de los métodos

Hasta el momento se han identificado clases, objetos y relaciones entre las clases y los objetos, pero aún no se ha concretado nada sobre cómo los objetos acceden a sus responsabilidades. Para ello se utilizará la notación:

```
<objeto>.<responsabilidad>
```

Utilizando dicha notación `alejandro` puede acceder a su atributo nombre de la siguiente forma:

```
alejandro.nombre
```

y al método `CepillarDientes` de la forma:

```
alejandro.CepillarDientes
```

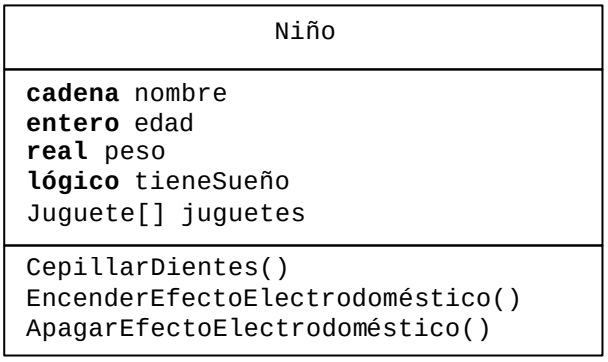
Nótese que tanto al atributo `nombre` como al método `CepillarDientes` se accede de forma semejante.

Pregunta: ¿Qué sucedería si se tuviera un atributo y un método con el mismo identificador?

Evidentemente no podría diferenciarse a cual de los dos se quiere acceder.

Una variante de solución a este problema es colocar un par de paréntesis ("`()`") al final de los métodos. Este convenio se utilizará en el desarrollo del capítulo.

En la siguiente figura se muestra un refinamiento de la clase `Niño` incluyendo el nuevo convenio.



Situación de análisis

Suponga que `alejandro` tiene un hermano (`carlos`) de apenas un año.

Pregunta: ¿Cómo `gloria` (madre de ambos niños) podría dormir a `alejandro` y a `carlos` utilizando el método `DormirNiño()`?

El método `DormirNiño()` debe ser capaz de dormir a cualquiera de los niños y por lo tanto debe tener la información de cual niño es al que se va dormir en un instante dado. El objeto `gloria` tendría que invocar al método `DormirNiño()` informándole cual de los niños es el que se va a dormir. Por ejemplo, si desea dormir a `carlos`:

```
gloria.DormirNiño(carlos).
```

Note que en la definición de la clase `Mamá` no queda explícito que el método `DormirNiño()` necesite alguna información.

Pregunta: ¿Cómo definir en la clase `Mamá` la información que el método `DormirNiño()` necesita?

De forma general esta información se coloca dentro de los paréntesis utilizando un identificador precedido de un tipo de dato como se muestra a continuación:

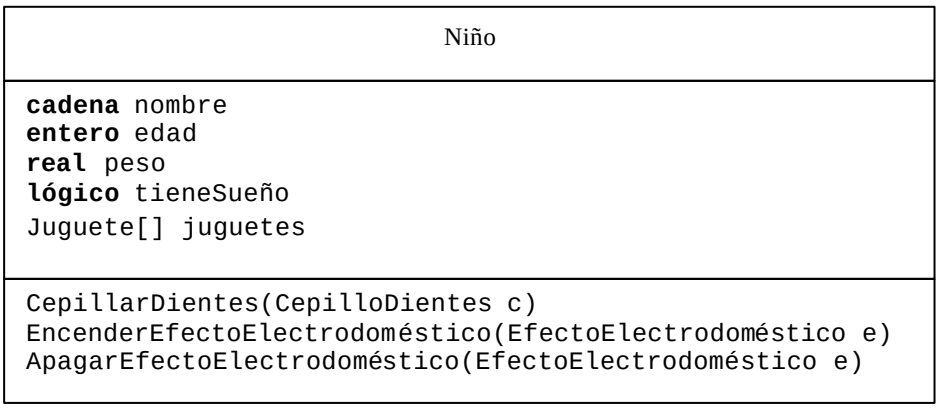
```
<identificador de método> ( <tipo de dato> <identificador> )
```

Por ejemplo, para el caso del método `DormirNiño()`:

```
DormirNiño(Niño unNiño)
```

La información que se le pasa a un método se denomina parámetro y en caso de ser más de uno, se separan por comas antecedidos de sus respectivos tipos de dato. No obstante, existen métodos que no necesitan parámetros y en su declaración mantienen los paréntesis vacíos (“`()`”).

En la siguiente figura se refina la clase `Niño`, añadiéndo los parámetros a los métodos (*declaración de métodos*).



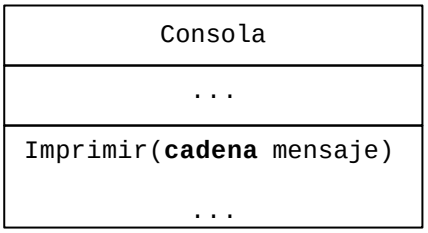
I.1.7 Encapsulamiento.

El acceso a las responsabilidades de los objetos tiene múltiples aristas. Por una parte el acceso a los métodos en principio siempre es de la misma forma:

```
<objeto>.<método>([<valores de los parámetros>])
```

Situación de análisis

Suponga la existencia de una clase `Consola` con un método `Imprimir` que permite mostrar por pantalla una cadena de caracteres como se muestra a continuación:



y se dispone de una instancia de la misma (`consola`).

Pregunta: ¿Cómo imprimir el nombre de `alejandro`?

```
consola.Imprimir(alejandro.nombre)
```

Otro es el caso cuando se habla de acceso a los atributos, pues se pueden usar sus valores como se mostró con anterioridad cuando se invocó un método del objeto `consola` para mostrar por pantalla el `nombre` de `alejandro`; o podríamos modificarlos como se muestra a continuación:

```
alejandro.nombre = "Jorge"
```

En este último caso se puede fácilmente perder la integridad del objeto propiamente dicho pues ahora a alejandro se le ha cambiado su nombre. De forma análoga podría modificarse la edad de alejandro de la siguiente forma:

```
alejandro.edad = -5
```

este valor es **`entero`** (-5) pero que no pertenece al dominio de valores del atributo `edad`.

A partir de las reflexiones anteriores es posible llegar a la conclusión de que es preciso disponer de mecanismos que permitan restringir el acceso a las componentes de los objetos para garantizar la integridad de los mismos, cobrando especial interés el concepto de encapsulamiento o encapsulación. A este mecanismo también se le denomina ocultación de la información y es la capacidad de ocultar los componentes internos de un objeto a sus usuarios y proporcionar una interfaz solamente para los componentes que se desee que el cliente tenga la posibilidad de manipular directamente [13]. El mecanismo de encapsulamiento o encapsulación será abordado con detenimiento en el tema II, específicamente en el capítulo II.3.

El mecanismo de encapsulamiento permite además, separar el qué del cómo, es decir, los objetos ofrecen a sus clientes una interfaz de qué es a lo que se puede acceder pero el cómo respecto a su funcionalidad interna queda oculto. Los procedimientos internos sobre cómo lleva a cabo un objeto un trabajo no son importantes mientras que ese objeto pueda desempeñar su trabajo.

I.1.8 Mensajes y métodos. Definición de Algoritmo

Con anterioridad, en más de una oportunidad se llegó a la conclusión de que las clases y los objetos no existen de forma aislada y esto permitió abordar el tema de las relaciones entre clases.

Un programa OO puede verse como un conjunto de objetos interactuando entre sí. El mecanismo de interacción entre los objetos es el mensaje, que puede entenderse como cualquier pedido o solicitud que se hace a un objeto para que ejecute una de las acciones predefinidas en él (un método). El mensaje especifica

qué acción realizar (y si es necesario algunos parámetros o argumentos), pero no cómo realizarla. Es decir, un programa OO es una secuencia de mensajes entre objetos para la solución de un problema.

Situación de análisis

Alrededor de las 9:00 de la noche, alejandro le dice a su mamá (gloria): tengo sueño. Inmediatamente gloria se ocupa de dormir al niño (también de darle a tomar leche como es habitual) y para ello debe llevar a cabo una serie de acciones o pasos (prepararle la leche, echarla en el pomo, cambiarle la ropa, acostarlo con la cabecita sobre la almohada, darle a tomar la leche e incluso acostarse a su lado hasta que se duerma).

Analizando la situación anterior se aprecia como interactúan los objetos gloria y alejandro a través del mensaje "tengo sueño". Este mensaje indica un pedido o solicitud que le hace alejandro a gloria para que ejecute el método `DormirNiño(…)`. Dicho mensaje especifica que acción realizar (`DormirNiño(…)`) y qué parámetro es necesario (alejandro). Finalmente, la forma que tiene el objeto *receptor* (gloria) para responder al mensaje ("tengo sueño") del objeto *emisor* (alejandro), es la siguiente:

```
gloria.DormirNiño(alejandro)
```

En la POO, la acción se inicia mediante la transmisión de un mensaje a un agente (un objeto) responsable de la acción. El mensaje tiene codificada la petición de una acción y se acompaña de cualquier información adicional (parámetros o argumentos) necesaria para llevar a cabo la petición. El receptor es el agente al cual se envía el mensaje. Si el receptor acepta el mensaje, acepta la responsabilidad de llevar a cabo la acción indicada. En respuesta a un mensaje, el receptor ejecutará algún método para satisfacer la petición. Todos los objetos de una clase dada usan el mismo método en respuesta a mensajes similares.

Una vez más aparece ante nosotros el concepto de encapsulamiento o principio de ocultación de la información, en esta oportunidad en la interacción entre los objetos a través del paso de mensajes; esto es, el cliente que envía una petición no necesita conocer el medio real con el que ésta será atendida.

Retomando la situación de análisis de la presente sección se aprecia como para dormir al niño, la mamá debe llevar a cabo una secuencia de pasos o acciones que tiene que ser organizada correctamente en el tiempo para cumplir su objetivo.

Es por ello que se puede definir para el método DormirNiño(Niño unNiño), que tiene como objetivo además de dormir al niño, darle a tomar un pomo de leche, las siguientes acciones:

```
preparar la leche
echar la leche en el pomo
cambiar de ropa al niño
acostarlo
colocar la cabecita sobre la almohada
darle a tomar la leche
acostarse a su lado hasta que se duerma
```

Esta secuencia de pasos no es única, pues sin dudas la siguiente secuencia de pasos también conduce al resultado deseado:

```
cambiar de ropa al niño
acostarlo
colocar la cabecita sobre la almohada
preparar la leche
echar la leche en el pomo
darle a tomar la leche
acostarse a su lado hasta que se duerma
```

Pregunta: ¿Qué características cumplen ambas secuencias de pasos que hacen posible que el niño se duerma?

- El número de acciones es finita.
- Todas las acciones son posibles de realizar, en este caso por gloria.
- El resultado esperado se obtiene en un tiempo finito.

Informalmente, una secuencia de pasos que cumpla con estas características recibe el nombre de *algoritmo*.

Sin embargo, no cualquier secuencia de pasos conduce al objetivo deseado, la siguiente secuencia es un ejemplo de ello:

```
cambiar de ropa al niño
acostarlo
colocar la cabecita sobre la almohada
acostarse a su lado hasta que se duerma
darle a tomar la leche
echar la leche en el pomo
preparar la leche
```

A través de esta secuencia de acciones es probable que se cumpla el objetivo de dormir al niño, pero resulta imposible que se tome la leche si se le da a tomar del pomo antes de echarla.

Existen disímiles definiciones de algoritmo , una posible definición de algoritmo es una secuencia finita de acciones, cada una de las cuales tiene un significado preciso y puede ejecutarse con una cantidad finita de esfuerzo en un tiempo finito.

En el modo tradicional un algoritmo especifica la secuencia de acciones que aplicada sobre un conjunto de informaciones produce el resultado deseado, por tanto, el proceso de creación de un algoritmo de forma tradicional consiste en seleccionar los conjuntos de informaciones y las acciones para entonces decidir cómo organizar las acciones en el tiempo [9]. En el caso del modelo de objetos se extiende el concepto de acciones e informaciones pues participan también las clases, los objetos, los mensajes y los métodos.

Para finalizar esta sección se presentan una serie de ideas muy didácticas que aparecen en [8] para definir algoritmo. "Informalmente se define un algoritmo como un conjunto de reglas, pasos u órdenes que indican una secuencia de operaciones a ejecutar para resolver un problema determinado que debe satisfacer los requisitos siguientes:

a. Ser finita. En caso de ser infinita la secuencia evidentemente no es útil para resolver ningún problema real.

b. Cada paso debe ser factible de realizarse por algún ser biológico o dispositivo. Por ejemplo, un algoritmo para viajar a la Luna no puede tener en sus pasos:
   Encienda el motor del automóvil.
   Arranque el automóvil.
   Póngalo en dirección a la Luna.
   Pise el acelerador y ruede hasta llegar a la Luna.
   porque evidentemente el último paso no es factible de realizar.

c. La ejecución de la secuencia de pasos debe dar un resultado en un tiempo finito. Por ejemplo, un algoritmo para subir una escalera no puede tener en su secuencia los pasos:
   1. *Suba un escalón.*
   2. *Baje un escalón.*
   3. *Vuelva al paso 1.*

   porque evidentemente aunque la secuencia es finita y cada paso es factible no dará una respuesta en un tiempo finito".

En la misma fuente, también se ejemplifica como un algoritmo correctamente diseñado el conocido algoritmo de Euclides para hallar el máximo común divisor (mcd) entre dos números naturales:

1. *Ponga el mayor de los dos números naturales en A y en B, el menor.*
2. *Divida el número A por el número B y deje el resto en R.*
3. *Si el número en R es igual a 0, entonces terminar (el mcd es el número que hay en B).*
4. *Ponga en A el número que está en B.*
5. *Ponga en B el número que está en R.*
6. *Vuelva al paso 2.*

Se le sugiere al lector ejecutar este algoritmo con números concretos.

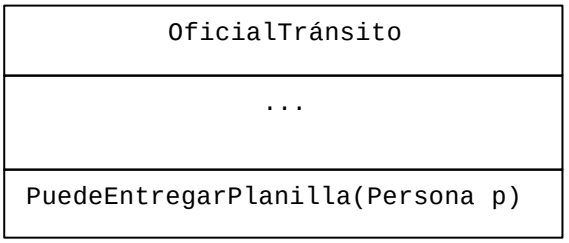
Situación de análisis

Para obtener la licencia de conducción, el padre del niño Alejandro se tuvo que presentar en la oficina de tránsito de su municipio y solicitar la planilla para la realización del examen médico. El oficial de tránsito Pablo López (`OficialTránsito pablo`) que fue quien lo atendió le preguntó la edad e inmediatamente que éste le respondió (21 años), el oficial le entregó la planilla y le dió algunas orientaciones.

Pregunta: ¿Cuál fue el criterio seguido por el oficial de tránsito pablo para entregarle la planilla al padre de Alejandro (Alexis)?

Simplemente determinar que `alexis` tenía más de 18 años que es el límite exigido para obtener la licencia de conducción. Note que para ello tuvo que llevar a cabo una secuencia de pasos entre los cuales se encuentra enviarle un mensaje al objeto `alexis` para solicitarle su edad y luego comparar la respuesta de éste (21) con 18 y comprobar que efectivamente es mayor (análisis de casos en II.1). A esta responsabilidad del `OficialTránsito pablo` (y de todas las instancias de la referida clase) se le denominará `PuedeEntregarPlanilla`.

Seguidamente se muestra una propuesta para la clase `OficialTránsito` y después se presenta como el objeto `pablo` utiliza la responsabilidad `PuedeEntregarPlanilla` para determinar si le puede entregar la planilla a `alexis`.




```
pedro.EntregarPlanilla(alexis)
```

Es de destacar que el parámetro del método PuedeEntregarPlanilla es de tipo Persona pues cualquier Persona puede solicitar dicha planilla. En lo particular alexis es una Persona.

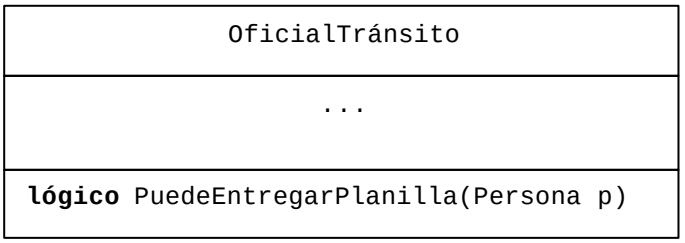
Situación de análisis

Evidentemente dentro del método `PuedeEntregarPlanilla` es donde se llevan a cabo los pasos o acciones para determinar si se le puede entregar la planilla o no a `alexis`.

Es significativo que este método tiene que devolver un valor (`si` ó `no`, `verdadero` ó `falso`).

Pregunta: ¿Cómo expresar que el método `PuedeEntregarPlanilla` devuelve un valor?

Esto se expresa a través del tipo de dato que tiene que retornar el método (**`lógico`**) delante del propio método como se muestra en la siguiente figura.



De forma general el tipo de retorno se coloca antes del identificador del método como se muestra a continuación:

```
<tipo de retorno > <identificador de método> ( [<parámetros>] )
```

I.1.9 Caso de estudio: `Selección de Atletas`

Situación de análisis

Imagínese una sesión de entrenamiento de la preselección nacional de atletismo donde se desea realizar pruebas de control a los atletas en las respectivas especialidades a las que estos pertenecen (carreras sin incluir los relevos), con el objetivo de conformar el equipo nacional.

Cada especialidad tiene un tiempo máximo requerido del cual tiene que bajar o igualar el atleta para integrar dicho equipo.

De los atletas se registrará el nombre, apellidos, carné de identidad, especialidad y tiempo marcado en el control. El tiempo se mide con un equipo eléctrico que funciona de manera similar a un cronómetro.

Un entrenador puede entrenar a más de un atleta, de los entrenadores se conoce el nombre y además son responsables de analizar la inclusión o no de los atletas en el equipo nacional de acuerdo al tiempo que hagan.

Ejercicio: Responder los siguientes incisos:

a) Identifique las clases envueltas en la situación planteada y sus responsabilidades.

b) Represéntelas mediante diagramas de clases, señalando las relaciones existentes.

c) Reconozca el mensaje que existe en la situación dada, quien lo emite y quien lo capta.

d) Diga una posible secuencia de pasos para determinar si un atleta pertenece o no al equipo nacional. ¿Se necesita de algún parámetro para determinar dicha secuencia?

Solución:

a) Identifique las clases envueltas en la situación planteada y sus responsabilidades.

Clases	Responsabilidades
`Preselección`	`Cjto de atletas`

`Atleta`	`nombre`
	`apellidos`
	`carné de identidad`
	`tiempo del control`
	`especialidad`
	`resultado de si integrara o no el equipo nacional`

`Especialidad`	`nombre` `tiempo máximo requerido`

`Tiempo`	`minuto`
	`segundo`
	`centésima`

`EquipoMedidorTiempo`	`tiempo para registrar las marcas`
	`Inicializarse a cero`
	`Arrancar`
	`Parar`
	`Mostrar el resultado`

`Entrenador`	`nombre`
	`cjto de atletas que entrena`
	`Analizar si un atleta puede integrar el equipo nacional`
	`Modificar el tiempo de un atleta después de la prueba`

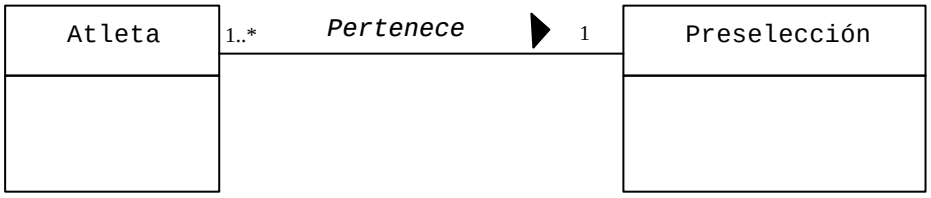
A continuación se resaltarán algunos detalles que pueden no quedar claros:

1. Cada atleta es de una especialidad determinada, pero a su vez las especialidades tienen un nombre por defecto y además un tiempo, lo que impide que puedan representarse por un tipo simple como `cadena`. Por lo tanto es necesaria una clase `Especialidad`.
2. Después de realizadas las pruebas, el atleta tendrá la información (muy importante para él) de si integrará o no el equipo nacional, y dicha información hay que representarla de alguna forma, de ahí la inclusión de esa responsabilidad en la clase.
3. La clase `Tiempo` se hace necesaria, debido a que muchas otras clases la usan, es el caso de `Atleta`, `EquipoMedidorTiempo` y `Especialidad`. Es más conveniente tener un atributo de tipo `Tiempo` que agrupe los datos minutos, segundos y centésimas, que tener a los mismos de forma independiente.
4. El mismo equipo de medir tiempo, es el encargado de medir todas las pruebas, por lo tanto debe ser capaz de inicializar los minutos, segundos y centésimas para cada prueba; arrancar al comienzo de una prueba, parar al final y mostrar el resultado de la misma.
5. Como un `Entrenador` puede entrenar a más de un `Atleta`, necesita tener como atributo el conjunto de dichos atletas.

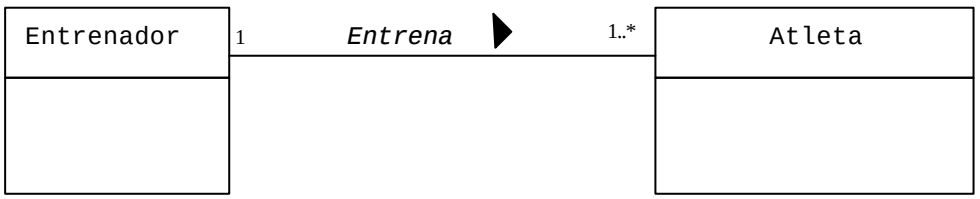
b) Represéntelas mediante diagramas de clases, señalando las relaciones existentes.

En el dominio del problema:

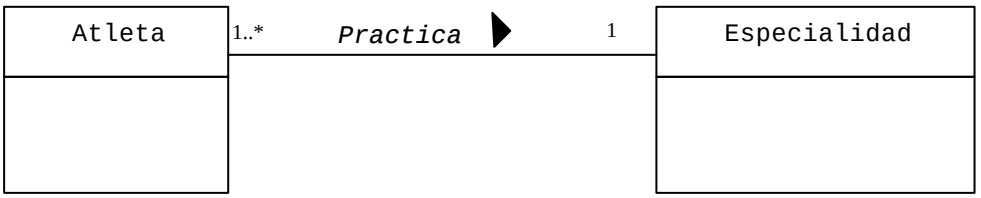
- Un atleta pertenece a una preselección, mientras a la preselección pertenecen muchos atletas. Aquí se aprecia una relación de asociación entre la clase `Preselección` y la clase `Atleta`.



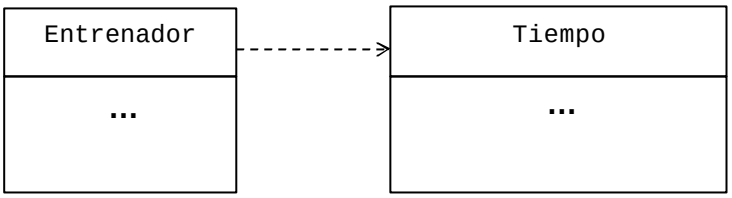
- Un entrenador entrena a uno o varios atletas, mientras que un atleta es entrenado por un solo entrenador. Esta es otra relación de asociación ahora entre las clases `Atleta` y `Entrenador`.



- Un atleta practica una especialidad, mientras que una especialidad puede ser practicada por varios atletas. Nuevamente se tiene una relación de asociación esta vez entre `Atleta` y `Especialidad.`



- El entrenador usa el resultado mostrado por el equipo medidor de tiempo para modificar el tiempo del atleta después de la prueba. Aquí se ve una relación de uso entre la clase `Entrenador` y la clase `Tiempo`.



- De igual forma la clase `Tiempo` es usada por la clase `EquipoMedidorTiempo` y la clase `Atleta.` Estas relaciones no se representarán por ser similares a la anterior.

c) Reconozca el mensaje que existe en la situación dada, quien lo emite y quien lo capta.

En la situación planteada hay un único mensaje que lo emite el equipo medidor de tiempo cada vez que termina una prueba, este mensaje es captado por el entrenador quien modifica el tiempo del atleta que corrió y posteriormente analiza su inclusión o no en el equipo nacional.

d) Diga una posible secuencia de pasos para determinar si un atleta pertenece o no al equipo nacional. ¿Se necesita de algún parámetro para determinar dicha secuencia?

Para determinar si un atleta puede pertenecer al equipo nacional se pueden seguir los siguientes pasos:

- Realizarle el control.
- Comparar el tiempo realizado por el atleta con el tiempo máximo que permite la especialidad.
- En caso de que este tiempo realizado sea menor o igual que el tiempo de la especialidad, entonces el atleta puede integrar el equipo nacional, en caso contrario no lo integrará.

Para llevar a cabo este algoritmo se necesita de dos parámetros que son el tiempo registrado por el atleta y el tiempo máximo permitido por la especialidad, pero tanto el tiempo registrado como la especialidad son atributos de la clase `Atleta`. Luego, solo se necesita pasar como parámetro al propio atleta.

I.1.10 Caso de estudio: `Universidad`

Situación de análisis

Suponga un día de clases en cualquier universidad de nuestro planeta y enunciemos entonces un conjunto de situaciones que se pueden dar en un intervalo de tiempo determinado:

a) El estudiante Juan Hernández del grupo Inf11 de la carrera de Ingeniería Informática, se dirige hacia el edificio de aulas y en el trayecto se encuentra con la secretaria docente de su facultad (Ana González), que le solicita pase un poco más tarde a verla para confrontar sus datos personales (número de identidad permanente, fecha de nacimiento y dirección particular) con los que se encuentran registrados en la secretaría de dicha facultad.

b) Poco antes de comenzar la clase de Introducción a la Programación la estudiante del grupo Inf11 Elsa Fernández, presidente de la Federación Estudiantil Universitaria (FEU) del referido grupo le informa a sus colegas de una propuesta de fechas para el desarrollo de los exámenes finales de las correspondientes asignaturas.

c) Ya en la clase de Introducción a la Programación el profesor Asistente, MSc. Ernesto Pérez le comenta a los estudiantes sobre los requerimientos del proyecto de curso y la fecha de entrega del mismo.

d) El también profesor Asistente, MSc. Pedro García solicita la información de los estudiantes con mejores promedios del grupo de segundo año (Inf21) y luego los de mejores resultados en la asignatura Programación I con el objetivo de seleccionar dos estudiantes para formar parte de la Casa de Software de la Facultad.

e) Al Decano de la Facultad, profesor Titular, Dr. Julio Herrera, el Rector de la universidad profesor Titular, Dr. Jorge Gómez le solicita la información del número de profesores jóvenes que pertenecen a la Facultad, el criterio de profesor joven es que tenga menos de 35 años.

Para desarrollar la situación de análisis anterior se seguirá una metodología similar a la utilizada para la situación de análisis de la sección I.1, con el objetivo de aplicar los conceptos que se presentan en este capítulo a la presente situación.

Ejemplo: Determinar algunos de los objetos y clases que aparecen en la situación de análisis presentada con anterioridad y algunas responsabilidades que aparezcan de forma explícita.

La solución del ejemplo se presenta en la siguiente tabla:

Clase	Objetos	Responsabilidades

`Estudiante`	`juan`
	`ana`
		`nombre`
		`apellido`
		`grupo`
		`especialidad`
		`númeroIdentidad`
		`fechaNacimiento`
		`direcciónParticular`

`Grupo`	`inf11` `inf21`	`identificación` `listaEstudiantes`
`SecretariaDocente`	`secretariaDocente`	`nombre` `apellido`
`Asignatura`	`introducciónProgramación`	`nombre`

`Profesor`	`ernesto`
	`pedro`
		`nombre`
		`apellido`
		`categoríaDocente`
		`categoríaCientífica`
		`asignaturaImparte`

`ProyectoCurso`	`proyectoIntPrograma ción`	`asignatura` `fechaEntrega`

`Decano`	`julio`	`nombre`
		`apellido`
		`categoríaDocente`
		`categoríaCientífica`

`Rector`	`jorge`	`nombre`
		`apellido`
		`categoríaDocente`
		`categoríaCientífica`

`Facultad`	`informática`	`nombre`
		`profesoresMiembros`
		`CantidadProfesoresJóvenes`

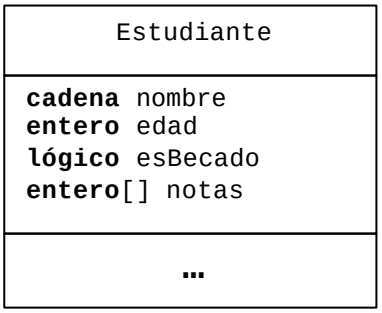
En un primer refinamiento es posible determinar algunas responsabilidades de los objetos hallados que no se muestran de forma explícita. En el inciso d) se precisa de los promedios de los estudiantes que componen el grupo y para calcular éstos se deben tener en cuenta las notas de cada una de las asignaturas aprobadas por los estudiantes del grupo, ello implica que es responsabilidad de los objetos `juan` y `ana` contener las notas de dichas asignaturas y el cálculo del promedio. En el inciso e) para obtener la cantidad de profesores con menos de 35 años entonces se necesita la determinación de la edad de éstos, luego también ésta podría pasar a ser una responsabilidad de los profesores `ernesto` y `pedro`. En el inciso a) se determina el objeto `secretariaDocente` y en el d) el objeto `decano`. ¿Tiene algún sentido ver estos objetos aisladamente? Claro que no, éstos son componentes del objeto `informática`, luego pasan a ser responsabilidades de este último objeto la información referida a la `secretariaDocente` y al `decano`.

Hasta aquí se ha identificado un grupo de objetos y clases, de los cuales se han determinado algunas de sus respectivas responsabilidades, claro está que si se continúa profundizando en el ejemplo, hay otros objetos presentes como la propia `universidad` que contendrá a `rector` y a las respectivas facultades. Los grupos tampoco pueden verse de forma aislada, éstos son responsabilidades de las respectivas facultades. Por

supuesto que en las facultades no solamente existe `decano`, `secretaria docente` y `profesores`, también existen trabajadores no docentes, vicedecanos, etc.

Ejemplo: Definir un diagrama de clases con una variante de la clase `Estudiante` donde se definan responsabilidades para el nombre, la edad, si es becado y notas de las asignaturas aprobadas. Clasifique los respectivos atributos

Es evidente que todas las responsanbilidades enumeradas pertenecen a las instancias de la clase `Estudiante` por lo que son atributos. Para obtener la representación de la clase apenas faltaría determinar el tipo de dato de cada uno de ellos. El nombre (**`cadena`**) y le edad (**`entero`**) ya fueron vistos en la clase `Niño`, faltando entonces un atributo para determinar *si es becado*, note que se presenta un atributo con dos estados posibles (`si` o `no`, `verdadero` o `falso`) por lo que el tipo **`lógico`** es el adecuado; por último en las `notas` se tiene un conjunto de valores enteros (**`entero`**`[]`).



Situación de análisis

En otra variante de la clase `Estudiante` es posible definir un atributo para la fecha de nacimiento que podría verse a su vez como una instancia de una clase `Fecha`.

Pregunta: ¿Qué tipo de relación se establece entre las clases `Estudiante` y `Fecha`?

Nótese que esta relación se establece en una sola dirección, en este caso la clase `Fecha` permite definir el atributo `fechaNacimiento` de la clase `Estudiante`. Esta relación se puede interpretar como que la clase `Estudiante` utiliza la clase `Fecha` en la definición de su atributo `fechaNacimiento`, por tanto, estamos en presencia de una relación de dependencia o uso.

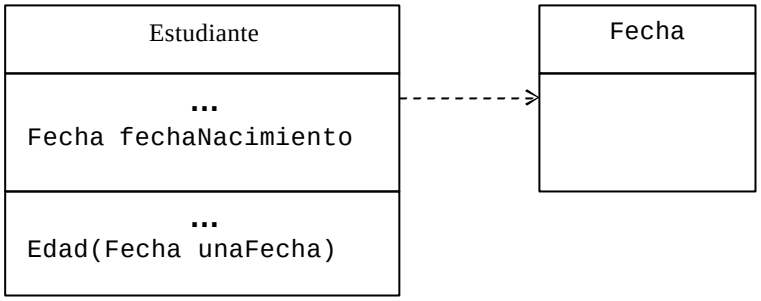
Pregunta: ¿Cómo es posible determinar la edad de un estudiante en una determinada fecha?

Para determinar la edad de un estudiante de esta forma, en primer lugar hay un dato "desconocido" que es "la fecha con respecto a la cual se desea calcular la edad a dicho estudiante" (parámetro). Evidentemente este parámetro es una instancia de tipo `Fecha`, incluso para calcular dicha edad el estudiante tendría que solicitarle el servicio a su fecha de nacimiento que determinará la diferencia con respecto a la nueva fecha.

Pregunta: ¿Qué tipo de relación se establece entre las clases `Estudiante` y `Fecha` a través del referido cálculo de la edad?

Nótese en este caso como la clase `Estudiante` tuvo que auxiliarse de la clase `Fecha` para definir el parámetro del cálculo de la edad, estableciéndose otra relación de uso entre las clases `Estudiante` y `Fecha`.

Ejemplo: Mostrar en un diagrama de clases la relación de dependencia o uso determinada entre `Estudiante` y `Fecha`.

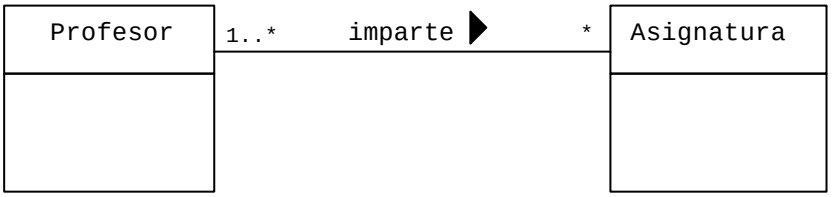


Situación de análisis

Todo `Profesor` puede impartir varias `asignaturas` (incluso ninguna), mientras que toda `asignatura` es impartida al menos por un `Profesor`.

Pregunta: ¿Qué tipo de relación se establece entre las clases `Profesor` y `Asignatura`?

Nótese que entre las clases `Profesor` y `Asignatura` existe una relación bidireccional donde todo objeto de la clase `Profesor` imparte varias asignaturas (incluso ninguna) y viceversa, toda `Asignatura` es impartida al menos por un `Profesor`. Estamos en presencia entonces de una relación de asociación que se representa en la siguiente figura:



Situación de análisis

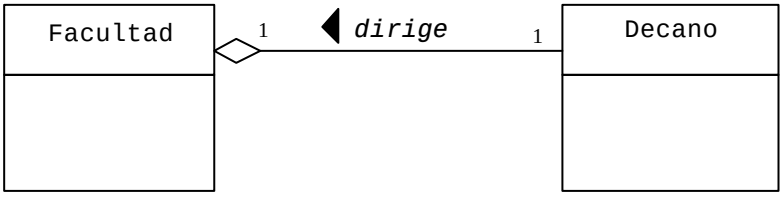
Toda `Facultad` es dirigida por un único `Decano`, mientras que a la vez un `Decano` puede dirigir solamente una `Facultad`.

Pregunta: ¿Qué tipo de relación se establece entre las clases `Decano` y `Facultad`?

Nótese que entre las clases `Decano` y `Facultad` también existe una relación bidireccional donde todo objeto de la clase `Decano` dirige una única `Facultad` y viceversa, toda `Facultad` es dirigida por un único `Decano`.

Este es otro ejemplo de relación de asociación que se establece ahora entre las clases `Facultad` y `Decano`. Una asociación normal entre dos clases representa una relación estructural entre iguales, es decir, ambas clases se encuentran conceptualmente en el mismo nivel, sin ser ninguna más importante que la otra [26]. Note que este no es el caso de Facultad y Decano pues este último es parte del primero. Esta relación todo/parte, en la cual una clase representa una cosa grande (el "todo") que consta de elementos mas pequeños (las "partes") es la que se denomina como *agregación* [26].

Una agregación representa una relación del tipo "tiene un", o sea, un objeto del todo tiene objetos de la parte; en realidad la agregación es solo un tipo especial de asociación y se especifica añadiendo a una asociación normal un rombo vacío en la parte del todo [26], como se muestra a continuación.

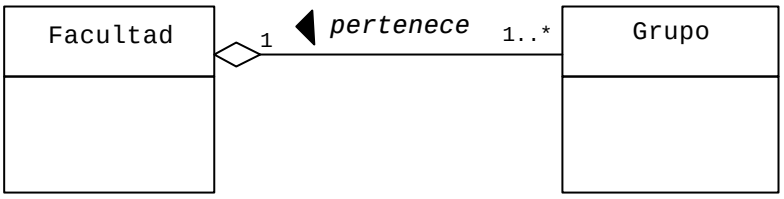


Situación de análisis

Todo `Grupo` de estudiantes pertenece a una única `Facultad` mientras que toda `Facultad` contiene varios `grupos` de estudiantes.

Pregunta: ¿Qué tipo de relación se establece entre las clases `Grupo` y `Facultad`?

Este es otro ejemplo de relación de agregación que se establece entre las clases `Facultad` y `Grupo`, esto está dado porque la clase `Grupo` ("parte") no tiene sentido al margen de la clase `Facultad` ("todo"). En la figura siguiente se muestra el diagrama de clases que representa la relación que se establece entre las clases Facultad y Grupo.

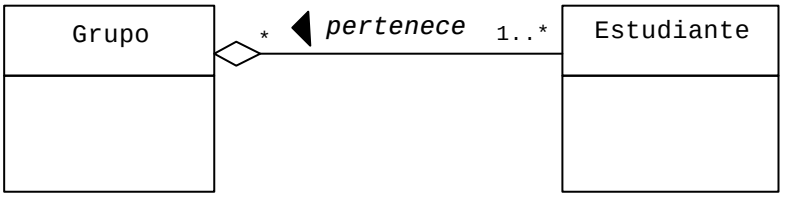


Situación de análisis

En un proceso docente definido a partir de créditos un mismo estudiante podría pertenecer a varios grupos en dependencia de las asignaturas que esté cursando, mientras que un grupo, evidentemente se compone de estudiantes.

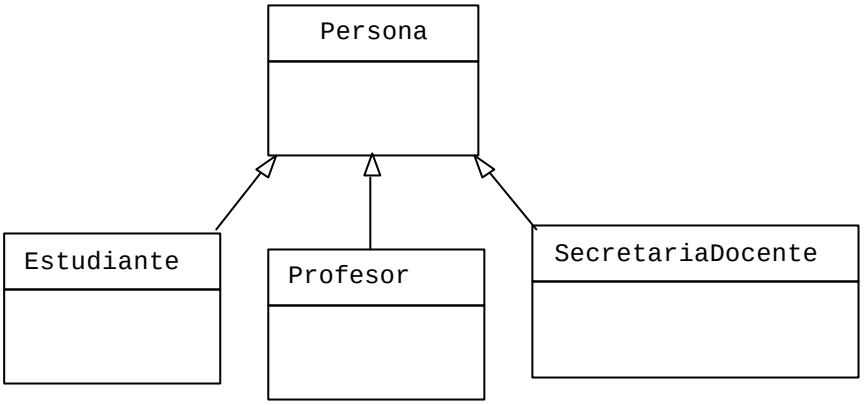
Pregunta: ¿Qué tipo de relación se establece entre las clases `Estudiante` y `Grupo`?

También este es un ejemplo de relación de agregación que se establece ahora entre las clases `Estudiante` ("parte") y `Grupo` ("todo"). A continuación se muestra el diagrama de clases que representa la relación que se establece entre estas clases.



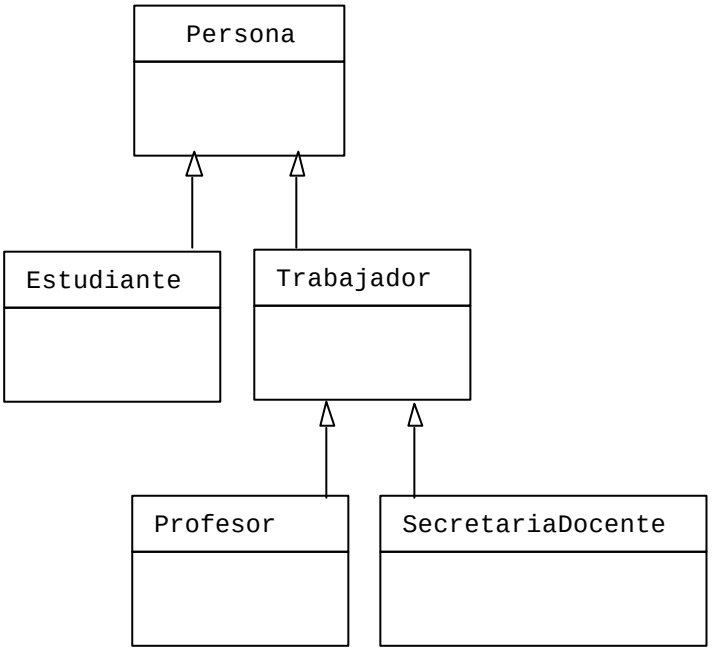
Pregunta: ¿Qué tipo de relación se puede establecer a partir de las clases `Estudiante`, `Profesor` y `SecretariaDocente`?

Si dudas que estas tres clases tienen un conjunto de responsabilidades comunes (por ejemplo, nombre, apellidos, fecha de nacimiento, edad, etc.). Incluso, es posible asegurar que estas tres clases se relacionan de la forma "es un" con respecto a una clase `Persona` donde se agrupan las responsabilidades comunes. Es decir, se puede establecer una relación de generalización como se muestra a continuación.



Pregunta: ¿Qué tipo de relación se puede establecer a partir de las clases `Profesor` y `SecretariaDocente`?

También es posible establecer una relación de generalización a través de la clase `Trabajador` que debe agrupar responsabilidades comunes como el salario por ejemplo. Note que en un dominio de problema donde los estudiantes no tengan ningún vínculo laboral no existe este tipo de relación con respecto a la clase `Estudiante`. A continuación se muestra un diagrama de clases donde se muestran estas relaciones.



Pregunta: ¿Cómo es posible imprimir por pantalla algunos de los datos del estudiante juan (auxiliándose del objeto `consola` referenciado en secciones anteriores?

Utilizando el método `Imprimir` de la clase `Consola` como se muestra a continuación:

```
consola.Imprimir(juan.nombre)
consola.Imprimir(juan.apellido)
consola.Imprimir(juan.direcciónParticular)
```

Pregunta: ¿Cómo el rector podría obtener la cantidad de profesores de la facultad de informática con menos de 35 años?

El objeto `rector` podría enviarle un mensaje al objeto `informática` y este responder de la siguiente forma:

```
informática.CantidadProfesoresJóvenes()
```

Situación de análisis

Analice las siguientes acciones:

```
juan.nombre = "Jorge"
juan.fechaNacimiento.día = 31
juan.fechaNacimiento.mes = 4
juan.año = 1981
```

Pregunta: ¿Qué ha sucedido con el objeto `juan`? ¿Cómo evitar este problema?

Sin dudas se ha perdido la integridad del objeto `juan` pues ahora éste ha cambiado su `nombre`. Por otra parte a `juan` se le modifica su atributo `fechaNacimiento` por el valor `31/04/1981`, que evidentemente no forma parte del dominio de objetos `Fecha`, de una vez el objeto `fechaNacimiento` (atributo de `juan`) ha perdido su integridad puesto que no existe instancia alguna con esos valores (estado) y luego `juan` también la ha perdido pues su fecha de nacimiento es incorrecta.

Para evitar este problema tenemos que limitar el acceso a las responsabilidades de `juan` a través del mecanismo de encapsulamiento que será estudiado más adelante.

Pregunta: ¿Cómo el decano de la facultad de informática podría obtener información respecto a alguno de sus profesores?

En este caso el decano de la facultad de informática podría enviarle un mensaje al objeto `informática` que es precisamente quien contiene la información de los profesores de la facultad. Para ello la clase `Facultad` (clase a la que pertenece `informática`) debe tener una responsabilidad que pueda satisfacer este mensaje, por ejemplo a través de un método `BuscaProfesor` que tendría un parámetro para el número de identidad del profesor a buscar y devolvería el profesor en cuestión como se muestra a continuación:

```
informática.BuscaProfesor(65080921481)
```

Seguidamente se presenta un refinamiento de la clase `Facultad`, donde entre otras cosas, se incluye el mencionado método.

Facultad
cadena nombre Decano decano SecretariaDocente secDocente Profesor[] profesoresMiembros ...
entero CantidadProfesoresJóvenes(); Profesor BuscaProfesor(**entero** númeroIdentidad) ...

I.1.11 Ejercicios

1. Para la situación de análisis que se presenta a continuación**,** realice un análisis similar a los que se desarrollan para las *situaciones de análisis* que se presentan a través del presente capitulo

Situación de análisis

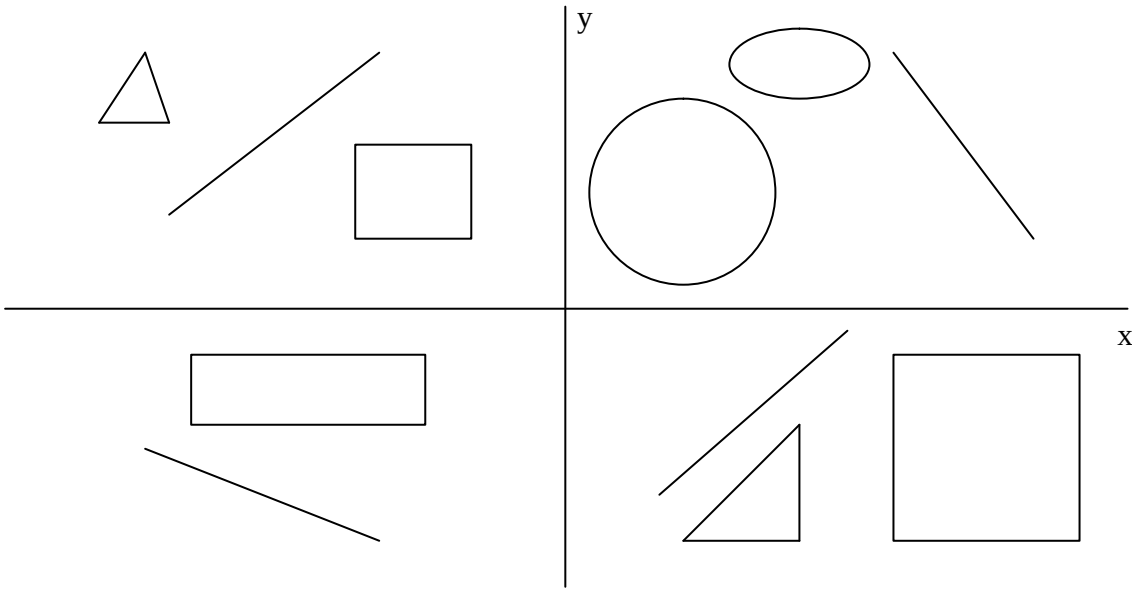
Imagine una tienda que se dedica a vender artículos de vestir, específicamente camisas, zapatos y pantalones. De la venta de los artículos se encarga el dependiente, persona que trabaja para la tienda y se ocupa de atender a los clientes. Particularmente se desea tratar de simular las acciones que realiza el dependiente de la tienda cuando se le solicita información de los artículos en venta.

En este caso el dependiente tiene que ser capaz de informar acerca del precio, talla, color y otras características (para las camisas mangas cortas o largas por ejemplo, para los zapatos si tienen cordones, etc.) de un determinado artículo que le soliciten los clientes.

2. Defina una situación de análisis a través de su experiencia como las que se definen en las *situaciones de análisis* anteriores. Posteriormente analice la situación de propuesta siguiendo una lógica similar a los desarrollos del capítulo.

3. Diseñe clases que permitan representar relojes, al menos tenga en cuenta que los relojes se ponen en hora y que mientras están funcionando avanzan hacia el instante siguiente transcurrido un segundo. Como un caso particular de estos relojes represente los relojes con alarma que implementan al menos una responsabilidad que permite determinar si puede sonar o no (idea original en [3]).

4. Diseñe clases para representar figuras geométricas del plano. Permita representar segmentos de recta, cuadrados, rectángulos, circunferencias, elipses y triángulos. Permita mover las figuras, calcular área, perímetro y/o longitud según sea el caso. Le sugerimos tener en cuenta que todas estas figuras geométricas pueden ser representadas a partir de un punto en común. Puede auxiliarse de la siguiente figura.



I.1.12 Bibliografía complementaria.

- Capítulos 1 y 2 de T. Budd**,** Introducción a la Programación Orientada a Objetos**,** Addison-Wesley Iberoamericana, 1994.

Capítulo I.2

I.2 Implementación de las clases en C#

En el capítulo anterior se presentaron los elementos básicos del modelo OO y para ello se centró la atención en la identificación de objetos, clases y algunas relaciones entre los mismos. Para la representación de las clases se utilizaron los diagramas de clases. El interés principal de este capítulo es implementar algunas de las clases identificadas y diseñadas en la sección anterior en un lenguaje de programación OO y al mismo tiempo presentar la *descomposición* como técnica básica en el diseño de algoritmos, facilitando el tratamiento de la complejidad y permitiendo a la vez organizar las acciones en el tiempo. Particularmente se presenta la *secuenciación* como técnica basada en la introducción de estados intermedios como forma de llegar del estado inicial al final.

En función de estos objetivos primeramente se debe seleccionar un lenguaje de programación que implemente lo mejor posible los conceptos del modelo OO. Los lenguajes de programación aíslan al programador de las complejidades inherentes al lenguaje de máquina de las computadoras. En el caso particular de los lenguajes OO posibilitan implementar clases e idear algoritmos con un alto nivel de abstracción.

Según Molina [6], "en la década pasada la POO se ha consolidado como el paradigma de programación más apropiado para construir software de calidad, sobre todo porque favorece la escritura de código reutilizable y extensible, además de reducir el salto semántico entre los modelos de análisis del problema y el código de la aplicación: los objetos del dominio del problema son objetos de la solución software".

Para Katrib [3], "programar en un lenguaje de programación OO es definir clases (nuevos tipos de dato) que "expresan" una determinada funcionalidad la cual es común a todos los individuos de una clase. A través de esta funcionalidad los objetos o instancias concretas de una clase dan respuesta a las solicitudes (mensajes) que les envían otros objetos".

Los beneficios y bondades de programar con un lenguaje OO son incalculables, esto puede apreciarse desde el punto de vista de la escritura de un código más eficiente, sobre todo por la posibilidad de reutilizar otros códigos así como por las facilidades de encapsular, modificar y extender dichos códigos; aunque quizás al principio esto no parezca exactamente así.

El lenguaje ideal para comenzar sería aquel que tuviese buenas propiedades, existiesen buenos entornos y abundante documentación, y que además su uso estuviese muy extendido entre las empresas de desarrollo.

A continuación se relacionan los elementos que caracterizan las buenas propiedades que deben tener los lenguajes OO [6]:

- Ser OO puro para obligar al usuario a programar dentro del paradigma, esto significa que toda entidad del dominio del problema se expresa a través del concepto de objeto.
- Reflejar con claridad los conceptos de la POO.
- Ser legible, pequeño y con gran potencia expresiva.
- Ser tipado estáticamente, para favorecer la legibilidad y fiabilidad.
- Permitir escribir asertos para posibilitar la programación por contratos (II.3).
- Incluir algún mecanismo de genericidad.

También se relacionan como exigencias para un buen entorno que sea robusto, gratuito, fácil de usar y que no necesite de muchos recursos.

Como es lógico no existe un lenguaje que cumple con todos los requisitos anteriores [6].

En la selección del lenguaje a utilizar se analizaron los seis más extendidos y utilizados en el ámbito académico: Object Pascal (Delphi), C++, Smalltalk, Eiffel, Java y C#.

Object Pascal y C++ fueron descartados por ser lenguajes híbridos pues ambos son descendientes de sus respectivos predecesores Pascal y C. Por las razones anteriores en Object Pascal y C++ se permite el uso de conceptos del paradigma imperativo procedural (híbrido).

Está demostrado que el uso de algunos recursos del paradigma imperativo-procedural pudiera ocasionar efectos dañinos y perjudiciales en la comprensión del modelo OO. En [20] se justifican estas razones.

En lo particular Object Pascal ha quedado muy distante en el mundo OO con respecto al éxito de Pascal (su predecesor) en la programación procedural. No es el caso de C++ que si tuvo gran aceptación tanto en el mundo empresarial como en el académico pero en la actualidad C++ se ha convertido en un lenguaje que lleva consigo una carga de viejas tecnologías (semántica de apuntadores por ejemplo). Por otra parte es un lenguaje "muy complicado y poco legible", elementos negativos al pensar en los principiantes.

Smalltalk tiene a su favor ser un lenguaje OO puro, muy simple pero de gran potencia expresiva y con buenos entornos de libre distribución que consumen pocos recursos pero es poco atractivo en este caso por tratarse de un lenguaje que no es tipado estáticamente [6].

En la misma fuente también se relacionan las buenas propiedades de Eiffel: OO puro, tipado, legible, incluye genericidad, mecanismo de asertos y un entorno robusto. Sin dudas, estas propiedades hacen considerar este lenguaje como una posible elección, sin embargo en la misma fuente se destaca que está muy poco extendido y no hay buenos entornos gratuitos, elementos que hacen que el mismo sea inapropiado teniendo en cuenta los objetivos previamente definidos.

Java ha sido reconocido por todos por su excelencia, muy extendido y utilizado en el desarrollo de aplicaciones Web y con gran popularidad en el ámbito académico. Además, existe abundante documentación y sobre todo está muy extendido en la industria, habiendo surgido alrededor de él importantes tecnologías para el desarrollo de aplicaciones OO.

Sin embargo, Java tiene algunas limitaciones como lenguaje OO puro. Específicamente estas se refieren al hecho de que Java tiene tipos primitivos y tipos objeto (estos conceptos se analizarán en detalle en I.2.2.1) que se tratan y se comportan de manera muy diferente. Además Java tiene en su contra las complicaciones de su primitiva de entrada System.In( ) y la falta de claridad de algunas construcciones que lo hacen inapropiado en este momento para nuestros objetivos. Debemos aceptar que si éstas fueran todas las opciones, Java sería el elegido pero aún nos resta una propuesta por analizar.

I.2.1 C#, nuestra elección

La elección ha sido C# por las razones que se presentan a continuación a partir del resumen de algunas de las ideas expuestas en [10].

Durante los últimos 20 años C y C++ han sido los lenguajes preferidos para desarrollar aplicaciones comerciales y de negocios. Sin embargo, cuando se comparan con lenguajes mucho menos potentes como Microsoft Visual Basic es apreciable que con C/C++ se necesita mucho más tiempo para desarrollar una aplicación. Muchos programadores de C/C++ se han limitado con la idea de cambiar de lenguaje porque podrían perder gran parte del control a bajo nivel al que estaban acostumbrados.

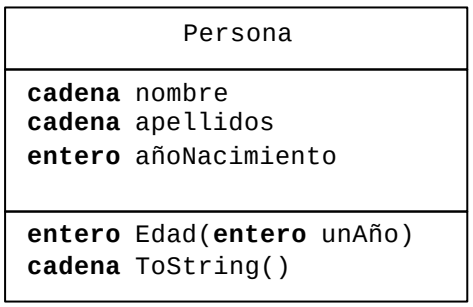
O sea, los programadores estaban necesitando un lenguaje entre los dos. Un lenguaje que ayudara a desarrollar aplicaciones de forma rápida pero que también permitiese un gran control y se integrase bien con el desarrollo de aplicaciones Web, XML y muchas de las tecnologías emergentes.

Facilitar un lenguaje sencillo de aprender para los programadores inexpertos es una de las ventajas de C# con respecto a otros lenguajes y que ha sido determinante en esta elección. Esto se complementa con la combinación en C# de las mejores ideas de lenguajes como C/C++, Delphi (Object Pascal) y Java con las mejoras de productividad de .NET Framework de Microsoft (Tema V). Esta decisión también se sustenta en C# por ser un lenguaje OO puro, que ha sido pensado "expresamente" para la plataforma .NET, por lo que entre otros elementos se dispone de una biblioteca de clases (BCL, Base Class Library) en la que existen aproximadamente 4000 clases ya definidas y disponibles.

I.2.2 Implementación de una clase en C#. Caso de estudio: clase Persona

Para ejemplificar la implementación de las clases en C# se toma como ejemplo la clase `Persona` desarrollada en el capítulo anterior. La propuesta de implementación de esta clase en C# se realizará a través

de una variante simplificada de la misma. En esta simplificación se sustituye la responsabilidad `Fecha fechaNacimiento` por la responsabilidad **`entero`** `añoNacimiento`. Esta simplificación se hace porque al expresar el atributo `Fecha fechaNacimiento`, se implementa una relación de uso entre las clases `Persona` y `Fecha` que será abordada y caracterizada en detalle en el capítulo II.2. Luego la clase `Persona` queda como se muestra en la figura siguiente:



Nótese que se insiste en la clasificación de los atributos, teniendo en cuenta el dominio (tipos de dato) al que pertenecen los valores a representar en cada uno de ellos.

El concepto de tipos de dato se define como un dominio de valores y un conjunto de operaciones sobre ese dominio. En todo lenguaje de programación deben existir tipos de dato básicos (simples, primarios o primitivos) para representar al menos enteros, reales, caracteres y lógicos. En la siguiente sección se presentan los tipos de dato básicos que ofrece C#.

I.2.2.1 Tipos de dato básicos

En la mayoría de los lenguajes se brinda un conjunto de tipos de dato que con mucha frecuencia son necesarios en la escritura de las aplicaciones, a éstos se les denominan tipos de dato básicos o primitivos y entre éstos no deben faltar los tipos de dato para representar enteros, reales, cadenas y lógicos. En la tabla siguiente se muestran estos tipos de dato básicos con su respectiva identificación en la BCL y el alias correspondiente.

Tipo	Descripción	Bits	Rango de valores	Alias
System.SByte	*Bytes* con signo	8	[-128, 127]	**sbyte**
System.Byte	*Bytes* sin signo	8	[0, 255]	**byte**
System.Int16	Enteros cortos con signo	16	[-32.768, 32.767]	**short**
System.UInt16	Enteros cortos sin signo	16	[0, 65.535]	**ushort**
System.Int32	Enteros normales	32	[-2.147.483.648, 2.147.483.647]	**int**
System.UInt32	Enteros normales sin signo	32	[0, 4.294.967.295]	**uint**
System.Int64	Enteros largos	64	[-9.223.372.036.854.775.808, 9.223.372.036.854.775.807]	**long**
System.UInt64	Enteros largos sin signo	64	[0,18.446.744.073.709.551.615]	**ulong**
System.Single	Reales con 7 dígitos de precisión	32	$[1.5\times10^{-45}, 3.4\times10^{38}]$	**float**
System.Double	Reales de 15-16 dígitos de precisión	64	$[5.0\times10^{-324}, 1.7\times10^{308}]$	**double**
System.Decimal	Reales de 28-29 dígitos de precisión	128	$[1.0\times10^{-28}, 7.9\times10^{28}]$	**decimal**
System.Boolean	Valores lógicos	32	**true**, **false**	**bool**
System.Char	Caracteres Unicode	16	['\u0000', '\uFFFF']	**char**
System.String	Cadenas de caracteres	Variable	El permitido por la memoria	**string**

Otro elemento muy importante a la hora de utilizar un tipo de dato es conocer cómo expresar los valores (constantes) o literales del dominio. A continuación se muestran algunos ejemplos de valores de los principales tipos de dato básicos.

Tipo de dato	Valores
int	1, -5, 532, -423
double	1.0, -5.0, 2.24, -43.54, 542.342
bool	**true, false**
char	'B', 'b', '@', '1', '/'
string	"Juan","Ejemplo de oración"

I.2.2.2 Sintaxis y semántica

Antes de ver como se implementa una clase se verán algunos conceptos y términos que serán muy útiles para comprender con mayor facilidad las estructuras que en lo adelante se abordaran en el texto.

Una clase en C# es una secuencia de símbolos (o caracteres) de un alfabeto básico. Esta secuencia de símbolos forma lo que se denomina el texto o código fuente de la clase. Sin embargo, no cualquier secuencia de símbolos de C# representa una clase (al igual que no cualquier secuencia de letras del alfabeto latino conforman una oración del idioma español) [8].

Hay dos aspectos que determinan si una secuencia de símbolos es correcta en C#: la sintaxis y la semántica.

Las reglas de sintaxis de C# son las que permiten determinar de que manera los símbolos del vocabulario pueden combinarse para escribir código que por su forma sea correcto. Así, por ejemplo, en el lenguaje natural son las reglas de la sintaxis las que nos permiten determinar que la siguiente secuencia

Caballo el el hombre cabalga sobre

no es correcta.

Nota: En este momento no se preocupe por el conjunto de reglas sintácticas de C# pues en el desarrollo del curso y del texto se abordaran la mayoría de ellas, en la medida que se presenten las diferentes estructuras sintácticas que ofrece C#.

La semántica por su parte guarda una estrecha relación con las acciones o instrucciones. Esta permite determinar el significado de la secuencia de símbolos para que se lleve a cabo la acción por la computadora. También a través de las reglas semánticas pueden ser detectados errores de interpretación que no permiten que las acciones o instrucciones puedan ser ejecutadas. Por ejemplo, aunque en el lenguaje natural la oración

El caballo cabalga sobre el hombre

es sintácticamente correcta, no lo es semánticamente pues no expresa una idea lógica. De forma similar son las reglas semánticas las que permiten determinar si una secuencia de símbolos no forma un código correcto, aunque no tenga errores sintácticos.

Para la descripción de las reglas sintácticas existe un formalismo conocido como Forma Normal de Backus o Forma de Backus y Naur que abreviadamente se denota por BNF (*Backus Normal Form* o *Backus Naur Form*). A través del texto, la sintaxis de los diferentes recursos de C# se irá describiendo con algunos elementos de este formalismo acompañándose de ejemplos y comentarios, de forma similar a como se realizó en el capítulo anterior.

Nota: No se usarán todos los elementos de la notación BNF, principalmente porque se tiene la opinión que complicarían a los lectores que tienen una primera relación con el apasionante mundo de la programación, hacia quienes prioritariamente está enfocado el presente texto.

Para la descripción de la semántica de los recursos de los lenguajes de programación no existe ningún tipo de formalismo por lo que éstos serán descritos mediante ejemplos y explicaciones.

I.2.2.3 Sintaxis para la definición de clases

Lo primero que hay que hacer para implementar clases en C# es precisamente conocer la sintaxis apropiada de una clase, la misma se muestra a continuación:

```
class <NombreClase>
```

```
{
      <miembros>
}
```

De este modo se define una clase con nombre <NombreClase> cuyos miembros son los que se definen en `<miembros>`, aunque en C# existen diferentes tipos de miembros por el momento se definirán apenas campos (atributos) y métodos. Los cuales estarán presentes en cada uno de los objetos o instancias de la clase.

Estos tipos de miembros (campos y métodos) se describen brevemente a continuación:

Campos: Son datos comunes a todos los objetos de una determinada clase y su sintaxis de definición es:

```
<TipoDato> <nombreCampo>;
```

El <nombreCampo> puede ser cualquier identificador que cumpla con las reglas establecidas que se presentan en I.2.2.6 y no coincida con el nombre de otro miembro previamente definido en la clase

Métodos: Son conjuntos de instrucciones a las que se le asocia un identificador de modo que si se desea ejecutarlas, basta referenciarlas a través de dicho identificador en vez de escribirlas.

La sintaxis que se usa para definir métodos es la siguiente:

```
<TipoDevuelto> <identificador> ([<Parámetros>])
{
      <instrucciones>
}
```

Dentro de estas <instrucciones> se puede acceder a todos los miembros definidos en la clase, a la cual pertenece el método. A su vez todo método puede devolver un objeto como resultado de haber ejecutado las <instrucciones>. En tal caso el tipo del objeto devuelto tiene que coincidir con el especificado en <TipoDevuelto>, y el método tiene que terminar con la cláusula return <objeto>; En caso que se desee definir un método que no devuelva objeto alguno se omite la cláusula return y se especifica void como <TipoDevuelto>.

Opcionalmente todo método puede recibir en cada llamada una lista de parámetros a los que podrá acceder en la ejecución de las <instrucciones> del mismo. En <Parámetros> se indican los tipos y nombres de estos parámetros y es mediante estos nombres con los que se deberá referirse a ellos en el cuerpo del método.

Nota: Verifique la coincidencia de estas definiciones con sus similares abordadas en el capítulo anterior.

Una vez que se tiene las herramientas imprescindibles para lograr una correcta implementación de clases en C# y partiendo de la representación de la clase Persona vista al inicio de la sección I.2.2 se verá que la traducción (implementación o codificación) a un lenguaje de programación es muy simple y de hecho existen herramientas que realizan esta traducción automáticamente.

```
// identificación de la clase
class Persona
{
      // atributos …
      // string es equivalente a cadena
      string nombre;
      string apellidos;
      // int es equivalente a entero
      int añoNacimiento;

      // método de consulta o acceso …
      int Edad(int unAño)
      {
            // edad aproximada
            return unAño - añoNacimiento;
      }

      string ToString()
```

```
        {
                return nombre + ”  ” + apellidos;
        }
}
```

Pregunta ¿Qué pasos se llevaron a cabo para realizar la traducción a C# de la clase `Persona`?

La implementación de la clase `Persona` a partir de su diagrama de clase se realiza apenas traduciendo cada una de las partes y conociendo por supuesto la sintaxis de definición de cada uno de los recursos implicados:

En primer lugar `<NombreClase>` se sustituye por `Persona`.

Posteriormente quedan por traducir los miembros de la clase, que resulta muy sencillo luego de haber tratado la sintaxis de los campos y métodos.

Dado que **`string`** es equivalente a **`cadena`** tanto que **`int`** es equivalente a **`entero`**, la traducción de los campos de acuerdo a su sintaxis queda:

```
string nombre;
string apellidos;
int añoNacimiento;
```

En el caso de los métodos es muy similar y se hace paso a paso, también a partir su sintaxis. Se toma como referencia el método `Edad` (`<identificador>`): como es un método cuya funcionalidad es informar de la edad de la persona a partir de su año de nacimiento y tomando como referencia un valor que significa el año en el cual se quiere conocer la edad de la persona, es seguro que se devolverá un valor por lo que `<TipoDevuelto>` tiene que ser especificado como **`int`** (equivalente a **`entero`**); así mismo se debe indicar que se devuelva un valor lo cual se hace mediante la cláusula

```
return unAño – añoNacimiento;
```

la cual relaciona en una expresión al parámetro `unAño` (valor tomado como referencia para el cálculo de la edad) y `añoNacimiento` (representa el campo que contiene el año de nacimiento de la persona) en una resta, con la cual se obtiene la edad aproximada de la supuesta persona. Las expresiones se abordaran en I.2.4.

Pregunta: ¿Cuál es el objetivo del método `ToString()`?

Este es un método que apenas transforma una instancia de la clase `Persona` en una cadena de caracteres, uniendo el nombre y los apellidos, este va a ser un método muy útil para cuando se desarrollen las aplicaciones.

De forma general este será un método que se definirá en casi todas las clases para mostrar un objeto en forma de cadena de caracteres.

I.2.2.4 Bloque de código

Pregunta: ¿Cómo se agrupan las responsabilidades de la clase Persona en la variante de implementación presentada?

En el empeño por “delimitar” las responsabilidades de la clase Persona y como parte de la sintaxis de definición de clases y métodos se utilizaron *bloques de código*: dígase de todos los elementos agrupados dentro de los símbolos “{” (apertura del bloque), inmediatamente después de la definición de y “}” (cierre del bloque).

En el caso particular de las clases, el bloque de código que agrupa las responsabilidades de la misma comienza inmediatamente después de `<NombreClase>`.

En C#, a diferencia de otros lenguajes, todo el código debe estar contenido en una clase, con escasas excepciones. Estas excepciones a la regla son la instrucción using (I.3), las declaraciones de estructuras (Parte II) y la declaración de namespaces (I.3). Cualquier intento de escribir código que no esté contenido en una

clase da como resultado un error de compilación. Por el momento se ha determinado las clases como el primer "contenedor" de un bloque de código.

Pregunta: ¿Cómo se agrupan las acciones u operaciones que se implementan en un método?, tome como caso de estudio el método `Edad` de la referida clase `Persona`.

Utilizando un bloque de código que comience inmediatamente después de (`<Parámetros>`) y termine después de la última instrucción que pertenece al método.

Se puede afirmar que los bloques de código permiten asociar código a las diferentes *estructuras gramaticales* del lenguaje. Las diferentes estructuras gramaticales del lenguaje se presentarán a lo largo del presente texto.

I.2.2.5 Análisis sintáctico y semántico

Como se había mencionado no cualquier secuencia de símbolos forman un código correcto en C# por lo que seguidamente se ponen a su consideración las siguientes situaciones:

Situación de análisis

Analice la secuencia de símbolos que se muestra en el siguiente listado:

```
clase Persona;
{
      cadena nombre;
      string apellidos;
      añoNacimiento int;

      int Edad(int unAño)
      {
            return unAño - añoNacimiento;
      }

      string ToString()
      {
            return nombre + " " + apellidos;
      }
}
```

Pregunta: ¿Qué errores sintácticos y/o semánticos se han cometido en el listado anterior?

Nótese que en la identificación de la clase se ha utilizado **`clase`** en vez de **`class`**, después de `Persona` se ha colocado un punto y coma (";"), **`cadena`** en vez de **`string`** delante de nombre y se invierten **`int`** y `añoNacimiento`.

Situación de análisis

Analice la secuencia de símbolos que se muestra en el listado siguiente:

```
class Persona
{
      string nombre;
      string apellidos;
      // bool es equivalente a lógico
      bool añoNacimiento;

      // método de consulta o acceso …
      int Edad(int unAño)
      {
            // edad aproximada
            return unAño - añoNacimiento;
      }
```

```
      string ToString()
      {
            return nombre + " " + apellidos;
      }
}
```

Pregunta: ¿Qué errores sintácticos y/o semánticos se han cometido en el listado anterior?

Nótese que el atributo `añoNacimiento` se ha definido de tipo **`bool`** (**`lógico`**). Este atributo toma parte en una resta dentro del método `Edad` (`unAño - añoNacimiento`) y éste ha sido definido como **`bool`**, sin embargo, las reglas semánticas (por defecto) de C# no permiten restar valores que no sean numéricos. Por lo tanto, aún cuando no hay errores sintácticos, el listado anterior no es correcto.

Los errores sintácticos y semánticas son detectados por el compilador de C#; sin embargo, este no puede hacer que una secuencia de acciones o instrucciones resuelva el problema para las cuales han sido diseñadas.

Situación de análisis

Analice la secuencia de símbolos que se muestra seguidamente

```
class Persona
{
      string nombre;
      string apellidos;
      int añoNacimiento;

      int Edad(int unAño)
      {
            return  añoNacimiento - unAño;
      }

      string ToString()
      {
            return nombre + " " + apellidos;
      }
}
```

Pregunta: ¿Qué errores sintácticos y/o semánticos se han cometido en el listado anterior?

Realmente no se han cometido errores sintácticos ni semánticos. No obstante, la clase implementada no cumple los requerimientos para los cuales fue diseñada. Nótese que la expresión para el cálculo de la edad (método `Edad`) ha sido alterada (`añoNacimiento - unAño`) con respecto a implementaciones anteriores. Al alterar la forma de calcular la edad (`añoNacimiento - unAño`) en el método `Edad`, evidentemente el resultado calculado no será el esperado, de hecho casi siempre el resultado va a ser un valor negativo, puesto que el valor que se toma como referencia para dicho cálculo siempre debe ser mayor que `añoNacimiento`.

Ahora estamos en presencia de una clase escrita correctamente desde el punto de vista sintáctico y semántico, por lo tanto el compilador de C# no detecta ningún error. Sin embargo, evidentemente esta clase no cumple con el objetivo para el cual fue diseñada.

Nota: Aún cuando los conceptos de sintaxis y semántica se han presentado en función del caso particular de C#, éstos son extensibles a cualquier lenguaje de programación, apenas haciendo cambios muy pequeños. De hecho la idea original para escribir estas secciones ha sido [8] y éste es un texto que se basa en otro lenguaje y paradigma.

I.2.2.6 Identificadores

Al analizar el código expuesto en el listado de la sección I.2.2.3 nótese que además de los diferentes símbolos utilizados para delimitar los elementos que en el intervienen ("{", "}", "(", ")" y ";") o definir operaciones ("-" y "+"), se han utilizado otros nombres simbólicos entre los cuales existen algunos resaltados en negrita (a

estas en particular se le denomina palabras reservadas del lenguaje y se presentarán en I.2.2.7) y otros que se han utilizado para representar las responsabilidades de la clase.

A estos nombres simbólicos se les denomina *identificadores*, los cuales se forman mediante una secuencia de letras y dígitos y en el caso de C# tienen que cumplir con las siguientes convenciones:

- El primer carácter de un identificador tiene que ser siempre una letra o un subrayado, el carácter subrayado se considera como una letra más.
- Un identificador no puede contener espacios en blanco ni otros caracteres distintos de los citados, como por ejemplo: * , : / ….

Los identificadores son utilizados para nombrar las diferentes entidades presentes en una clase en C#. En el caso de C# también se debe tener cuidado porque es un lenguaje *case sensitive* por lo que identificadores que apenas se diferencian en una mayúscula ya son diferentes (Color <> color) y además para los identificadores se utilizan caracteres *unicode* por lo que son identificadores válidos tanto año como maría.

Situación de análisis

Analice las secuencias de símbolos que se muestran a continuación:

```
nombre
Pedro Pérez
añoNacimiento
99
Edad
Edad_Promedio
_Valor1
Mínimo%
ToString
12abcd
xxx1
```

Pregunta: ¿Determine cuales de las secuencias no constituyen identificadores? Justifique.

Pedro Pérez	No puede haber espacio dentro de un identificador. Esto será considerado como dos identificadores
99	No empieza con letra o subrayado
Mínimo%	El símbolo % no es letra, número o subrayado
12abcd	No empieza con letra o subrayado

Algunos principios importantes sobre el uso de los identificadores que deben tenerse en cuenta [8]:

- Dos identificadores consecutivos deben estar separados por al menos un espacio, tabulación, cambio de línea o comentario.
- Usar identificadores nemotécnicos (que sugieren el significado de la entidad que nombran). Por ejemplo, en el listado de la sección I.2.2.3 nombre, apellidos y Edad son mas significativos para la lectura que n, a y E.
- No sacrificar la legibilidad por la longitud de los identificadores. Utilizar identificadores pronunciables. Esto ayuda a la memorización de los nombres y a la explicación oral del código fuente. No utilizar aN en lugar de añoNacimiento.

I.2.2.7 Palabras reservadas

Pregunta: ¿Cuál es el objetivo de los identificadores **`class`** y **`string`** en la clase `Persona` implementada en el listado de I.2.2.3?

En el caso de **`class`** su objetivo es informarle al compilador que el identificador `Persona` describe una clase; mientras que **`string`** permite “informar” que los atributos nombre y apellidos almacenarán

cadenas de caracteres, además **`string`** también permite informar que el método `ToString` va a retornar una cadena de caracteres.

Algunos identificadores tienen un uso especial en los lenguajes de programación. Estos identificadores son conocidos como palabras claves o palabras reservadas (*key words* o *reserved words*), ya que el programador no puede darle un uso diferente al que el lenguaje les confiere. Aunque ellos también se describen por una secuencia de letras son considerados como símbolos del vocabulario. Para destacarlos del resto de los identificadores serán denotados en negrita.

En C# se definen los siguientes identificadores como palabras reservadas:

`abstract, as, base, bool, break, byte, case, catch, char, checked, class, const, continue, decimal, default, delegate, do, double, else, enum, event, explicit, extern, false, finally, fixed, float, for, foreach, goto, if, implicit, in, int, interface, internal, lock, is, long, namespace, new, null, object, operator, out, override, params, private, protected, public, readonly, ref, return, sbyte, sealed, short, sizeof, stackalloc, static, string, struct, switch, this, throw, true, try, typeof, uint, ulong, unchecked, unsafe, ushort, using, virtual, void, while`

Nota: En este momento no se preocupe por el significado y uso de cada uno de estos símbolos, en la medida que se avance sin dudas que llegará a familiarizarse con la gran mayoría de ellos. De hecho ya se han utilizado algunos, por ejemplo, **`class, string, int`**, etc.

I.2.2.8 Comentarios

Hacer uso de una indentación y espaciamiento adecuados, así como una selección cuidadosa de los identificadores puede hacer legible y autoexplicativo un código fuente. Sin embargo, puede ser conveniente incluir en los códigos fuentes textos adicionales que expliquen o comenten determinada parte de este.

Analizando una vez más el listado de I.2.2.3 se pueden apreciar construcciones gramaticales o estructuras sintácticas que aún faltan por analizar: los *comentarios*. Por ejemplo:

```
// identificación de la clase
// atributos …
// string es equivalente a cadena
// int es equivalente a entero
// método de consulta o acceso …
```

Un comentario es texto que se incluye en el código fuente con el objetivo de facilitar su legibilidad a los programadores. Los comentarios no tienen significado alguno para la ejecución de una aplicación; esto es equivalente a decir que los comentarios son completamente ignorados por el compilador.

Los comentarios suelen usarse para incluir información sobre el autor del código, para aclarar el significado o el porqué de de determinadas secciones de código, para describir el funcionamiento de los métodos, etc.

En C# hay dos formas de escribir comentarios. Dado que muchas veces los comentarios que se escriben son muy cortos y no suelen ocupar más de una línea, C# ofrece una sintaxis compacta para la escritura de este tipo de comentarios en la 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 ejemplos de su uso aparecen en el listado de I.2.2.3:

```
// identificación de la clase
// atributos …
// string es equivalente a cadena
```

La segunda se utiliza para comentarios que abarquen más de una línea y 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 escriben
   comentarios que ocupen varias líneas */
```

En la práctica los programadores crean sus propios estilos para utilizar y escribir los comentarios, sin embargo, deben tenerse en cuenta los siguientes consejos [8]: no utilizar comentarios redundantes donde el propio texto del programa es lo suficientemente explícito y claro, ni tratar de resolver con comentarios los errores de los códigos, ni tratar de aclarar con éstos malas y confusas programaciones de los métodos.

I.2.3 Declaración de variables

Pregunta: ¿Cuál es el objetivo de los campos `nombre`, `apellidos` y `añoNacimiento` en la clase `Persona` implementada en I.2.2.3?

El objetivo de estos campos es conservar valores de algunas responsabilidades de las instancias u objetos de la clase `Persona` (nombre, apellidos y año de nacimiento respectivamente) durante la ejecución de las aplicaciones.

Las *variables* son entidades de los programas que se utilizan para conservar valores durante la ejecución de éstos. Las variables permiten representar valores que no necesariamente se conocen cuando se escriben las aplicaciones y a su vez pueden cambiar durante la ejecución de éstas. Al contenido de las variables se tiene acceso a través de los nombres simbólicos (identificadores) que las identifican.

Una variable es una abstracción de una zona de memoria que se utiliza para representar y conservar valores de un determinado tipo.

De lo anterior se deduce que una variable debe tener un tipo asociado que es precisamente el tipo de los valores que ella puede conservar. Esta definición de las variables que dice cuáles identificadores serán utilizados como nombres de variables y cual es el tipo asociado a estas, es lo que se denomina *declaración de variables*.

Ejemplo: Analizar nuevamente el listado de I.2.2.3 y determinar cuáles de las estructuras sintácticas podrían ser consideradas como *declaración de variable*.

Es preciso recordar que en una declaración de variable debe existir una asociación entre un identificador de una variable y su respectivo tipo.

Por otra parte, los atributos (como concepto del modelo OO) pueden ser considerados como un caso particular de variable teniendo en cuenta que éstos pueden conservar valores durante la ejecución de las aplicaciones. De hecho, en C# se definen estructuras sintácticas muy particulares y específicas que se les denomina atributo (que no serán objeto de análisis en el presente texto); es por ello que a partir de ahora a los atributos del modelo OO se les llamará campos o *variables de instancia* para no traer confusiones futuras con los atributos propios de C#.

También se pueden definir variables dentro del cuerpo de un método, las cuales se pueden calificar como *variables locales*, pues solo podrán ser accedidas dentro del cuerpo del método, algo similar sucede con las variables que constituyen parámetros de un método. Sobre los elementos relacionados con los métodos se profundizará en I.2.2.5.

Por supuesto que existen otros tipos de variables que se analizarán a lo largo del texto.

Teniendo en cuenta los elementos relacionados con anterioridad es posible determinar como declaración de variable en el mencionado listado a los siguientes ejemplos:

```
string nombre;
string apellidos;
int añoNacimiento;
int unAño;
```

Las variables son caracterizadas por un tipo (`int`) y un nombre simbólico (`añoNacimiento`). El tipo determina la clase de valores que puede tomar una variable (valores enteros en el caso de `añoNacimiento`), así como las operaciones en las que puede participar (aritméticas en `añoNacimiento`). Una buena práctica en programación es dar a las variables nombres que evoquen el objeto que representan (nombres nemotécnicos).

El conjunto de valores de todas las variables (objetos) en un instante dado se le denomina estado. Lógicamente, el estado de los objetos cambia cada vez que se ejecuta una acción que modifica el valor de al menos uno de sus campos.

A las variables de C# se les asigna un tipo, que es una descripción del tipo de datos que va a contener dicha variable. Como reglas en la definición y uso de las variables en C# se tiene que todas las variables tienen que ser definidas antes de ser utilizadas y que éstas pueden ser definidas en cualquier parte del código. Por otra parte, toda variable tiene que ser definida una única vez en un mismo bloque o segmento de código y a partir de esta definición puede ser utilizada todas las veces que se desee.

Las variables son declaradas escribiendo su tipo, seguido de algún espacio en blanco y luego el identificador de la misma terminando con un punto y coma. En caso de que se desee definir más de una variable entonces los identificadores de las mismas tienen que ser separados por coma. Al declarar una variable se está informando al compilador de C# de su tipo y de su identificador propiamente.

La estructura sintáctica para la declaración de variables se podría generalizar de la siguiente forma:

```
<tipo de dato> <lista de identificadores de variables>;
```

En el siguiente listado se muestra un uso de esta sintaxis para la declaración de variables:

```
class Persona
{
      string nombre, apellidos;
      int añoNacimiento;

      int Edad(int unAño)
      {
            return unAño - añoNacimiento;
      }

      string ToString()
      {
            return nombre + " " + apellidos;
      }
}
```

Para finalizar esta sección, en el siguiente listado se presentan otros ejemplos de definiciones de variables dentro de un bloque de código:

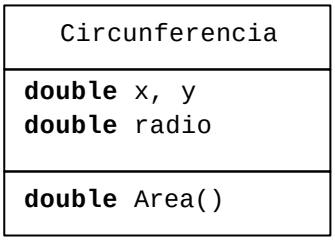
```
{
      int edad;
      uint _Valor1;
      double peso;
      string nombre1, nombre2;
      ...
}
```

I.2.3.1 Constantes. Caso de estudio: clase `Circunferencia`

Ejemplo: Diseñar una clase para representar circunferencias.

Para diseñar la clase se sugiere comenzar por determinar cuáles podrían ser las responsabilidades a tener en cuenta. En el caso de las circunferencias se debe tener en cuenta al menos responsabilidades para el centro, radio y el cálculo del área. En un segundo momento se clasifican dichas responsabilidades

teniendo entonces al centro (coordenadas “x” y “y”) y el radio como campos y al área como método como se muestra a continuación:



Pregunta: ¿Qué elementos se han tenido en cuenta para la clasificación de las responsabilidades de la clase `Circunferencia` en variables o métodos?

Para esta clasificación se ha tenido en cuenta que tanto las coordenadas del centro como el radio son informaciones inherentes a las instancias de la referida clase, mientras que el área es un valor que se calcula multiplicando el valor 3.14159 (π) por el cuadrado del radio.

Ejemplo: Implementar la clase `Circunferencia` en C#.

Siguiendo la misma metodología que fue utilizada en la implementación de la clase `Persona`, para implementar la clase `Circunferencia` primero se traduce el diagrama de la figura anterior y luego se implementan las acciones del método `Area` como se muestra a continuación.

```
clase Circunferencia
{
      double x, y;
      double radio;

      double Area()
      {
            return 3.14159 * radio * radio;
      }
}
```

Situación de análisis

Analice el listado anterior, note el uso del valor 3.14159 en la expresión para calcular el área. Sin dudas este es un valor que es posible necesitar en otras operaciones. Tómese por ejemplo que se desean ampliar las responsabilidades de la clase `Circunferencia` implementada anteriormente incluyendo un método `Longitud` para permitir de esta forma que los objetos de la referida clase puedan ser capaces de calcular además de su área, su longitud. Una variante de implementación de dicho método sería:

```
      double Longitud()
      {
            return 2 * 3.14159 * radio;
      }
```

Utilizar explícitamente este valor en las operaciones puede frecuentemente conducir a errores pues cuando se tienen algunas líneas de código puede ser prácticamente imperceptible la sustitución de este valor por 3.14169 por ejemplo y entonces los resultados distarían de los esperados. Nótese que en este caso se está hablando de un dato que siempre tiene el mismo valor, salvando la excepción de cambios en el grado de precisión.

Pregunta: ¿Existe en C# algún recurso sintáctico para tratar estas situaciones?

C# ofrece la declaración de *constantes* como recurso sintáctico para representar datos que no cambian durante toda la aplicación. A las constantes también se les asocia un tipo de datos. Una constante también puede ser definida como una variable cuyo valor es determinado durante el proceso de compilación.

Las constantes se definen como variables normales, ya sean campos o variables locales (exceptuando los parámetros) pero precediendo el nombre de su tipo la palabra reservada **`const`** y dándoles siempre un valor inicial al declararlas. Es decir, con esta sintaxis:

```
const <tipo de dato> <identificador> = <valor>;
```

Ejemplo de definición de constante:

```
const double PI = 3.14159;
```

Como C# es un lenguaje orientado a objetos puro, solamente recursos muy excepcionales se expresan fuera de una clase y este no es el caso de las constantes. Particularmente en la `BCL` existe una clase denominada `Math` en la cual se define la constante π(PI).

Ejemplo: Teniendo en cuenta los elementos anteriores, redefinir el método `Area`.

En este caso solo hay que sustituir el valor 3.1416 por el de la constante PI de la clase `Math` (`Math.PI`) como se muestra seguidamente:

```
double Area()
{
    return Math.PI * radio * radio;
}
```

Hay que destacar que en este ejemplo se accede a `PI` a través del nombre de la clase de la forma `Math.PI` y no a través de un objeto como se había especificado desde el capitulo I.1, la justificación de este acceso es el hecho de que `PI` es una constante y estas son implícitamente estáticas o sea *variables de clase* (II.2.2)

Considere nuevamente una aplicación donde se hace uso reiterado de la constante matemática π para mostrar algunas de las ventajas que reportaría el uso de la constante `Math.PI`:

- Es tedioso y monótono escribir constantemente la secuencia 3.14159 en vez de `Math.PI`, además como se expuso con anterioridad se pueden cometer errores que no serían detectados por el compilador.
- Mayor legibilidad para los códigos fuentes.
- Si se decide cambiar la precisión (usar por ejemplo 3.1415926) se tendrían que hacer modificaciones en todos los lugares donde aparece la secuencia de dígitos.

Ejemplo: La última de las ventajas expuestas con anterioridad trae consigo una problemática: ¿Cómo modificar una constante de una clase de la `BCL`?

Esto no es posible pues en principio no se debe contar con el código fuente de la clase `Math` de la `BCL` para cambiar el valor de PI, pero esto se podría solucionar definiéndose una constante con nombre PI en la propia clase circunferencia como se muestra a continuación:

```
clase Circunferencia
{
    double x, y;
    double radio;

    const double PI = 3.1415926;

    double Area()
    {
          return PI * radio * radio;
    }

    double Longitud()
    {
          return 2 * PI * radio;
```

```
    }
}
```

I.2.4 Expresiones y Operadores

Un *operador* es un carácter o grupo de caracteres que actúa sobre uno, dos o más *operandos* para realizar una determinada operación con un determinado resultado. Los operadores pueden ser unarios, binarios y ternarios según actúen uno, dos o tres operandos respectivamente. Llamaremos *operandos* a un valor constante o literal, al contenido de una variable (objeto), al resultado de la invocación de un método, al resultado de la evaluación de una operación, etc.

A continuación se describen algunos 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 muestran los usos más comunes de los mismos:

- Los *operadores aritméticos* incluidos en C# son los clásicos de *suma* (+), *resta* (-), *producto* (*), *división* (/) y *módulo* (%). También se incluyen operadores de *menos unario* (-) y *más unario* (+).
- Los *operadores lógicos* permiten realizar las operaciones lógicas típicas: *and* (&& y &), *or* (|| y |), *not* (!) y *xor* (^).
- Como *operadores relacionales* se han incluido los tradicionales operadores de *igualdad* (==), *desigualdad* (!=), *mayor que* (>), *menor que* (<), *mayor o igual que* (>=) y *menor o igual que* (<=).
- Para la 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, nótese en el listado de la sección I.2.2.3 como el método `ToString()` para retornar el nombre y a continuación los apellidos de los objetos Persona se auxilia de la expresión nombre + " " + apellidos.

Seguidamente se muestran los tipos de dato básicos y las operaciones que se pueden realizar sobre ellos:

Tipo de dato	Operaciones
`int`	+, -, *, /, % : **int** x **int** → **int** <, >, == , <=, >=, !=: **int** x **int** → **bool**
`double`	+, -, *, / : **double** x **double** → **double** <, >, == , <=, >=, !=: **double** x **double** → **bool**
`bool`	&, | : **bool** x **bool** → **bool** no : **bool** → **bool**
`char`	<, >, == , <=, >=, != : **char** x **char** → **bool** **+** : **char** x **char** → **int**
`string`	== , != : **string** x **string** → **bool** **+** : **string** x **string** → **string**

Pregunta: ¿Cómo se puede obtener el valor exacto de la división de 5 entre 2?.

En este caso se tienen dos valores enteros, por lo tanto el resultado de la división 5 / 2 es un valor entero (2) como se especificó en la tabla anterior.

Para obtener el valor exacto, simplemente se tiene que expresar al menos uno de los valores como constante del dominio de los números reales: 5.0 / 2 (5 / 2.0 ó 5.0 / 2.0).

Ejemplo: Clasificar los elementos que intervienen en el cálculo de la edad en el listado de I.2.2.3 (método `Edad`).

Recordemos que el referido cálculo se realiza a través de la *expresión*

```
unAño – añoNacimiento
```

en la cual intervienen dos *operandos* (`unAño` y `añoNacimiento`) y un *operador* ("-").

Denominamos *expresión* a una combinación de *operaciones* sobre *operandos* (valores) que nos permite obtener nuevos valores. Una *expresión* también puede ser definida como un operando o la aplicación de operadores sobre *operandos*.

La expresión es equivalente al resultado que proporciona el aplicar sus operadores a sus operandos. Una expresión puede estar formada por otras expresiones más sencillas, y puede contener paréntesis de varios niveles agrupando distintos términos.

I.2.4.1 Operador de asignación

Teniendo la riqueza expresiva del *operador de asignación* "=", se ha dedicado esta sección para su análisis particular.

En C# para asignarle valores a las variables es utilizado el *operador de asignación* "=", cuya sintaxis es:

```
<variable> = <expresión>;
```

La instrucción de asignación es un caso particular de instrucción que permite asignarle ("=") a una variable (`<variable>`) el resultado de la evaluación de una expresión (`<expresión>`), siendo particularmente para la OO el mecanismo utilizado para el cambio de estados entre los objetos.

A continuación se muestran una serie de definiciones y asignaciones a variables que pertenecen a los tipos de dato básicos de C#.

```
{
     int temperatura;
     uint edadJuan;
     temperatura = -5;
     uint edadPedro = 23;
     edadJuan = edadJuan + 2;
     edadPedro = edadPedro + 1;
     double peso = 60.5;
     char letra = 'a';
     string nombreJuan = "Juan";
     string  nombrePedro, nombreAna;
     nombrePedro = nombreJuan;
     nombreAna = "Ana";
     bool existe = true;
     ...
}
```

La instrucción de asignación se ejecuta cumpliendo el siguiente principio: "se evalúa la expresión a la derecha del operador de asignación (=) y el resultado de esta evaluación se le da como valor a la variable que está a la izquierda del operador". Luego de haberse efectuado esta asignación de valores a la variable de la izquierda este valor sustituye cualquier otro que hubiera tenido con anterioridad la variable.

Nota: Es importante diferenciar este operador de asignación con el utilizado en las igualdades matemáticas.

Nótese que en C# es posible relacionar la propia definición de las variables con el primer valor que se le asigna, en este caso se le llama inicialización. En C# existe distinción entre las variables *asignadas* y *no asignadas*. Se entiende por *variables asignadas* aquellas que han recibido un valor en algún momento del código y las *variables no asignadas* a aquellas que no han recibido valor alguno. Luego, en C# no es posible utilizar *variables no asignadas* porque su valor es desconocido y esto podría generar errores durante el proceso de compilación de su código (el proceso de compilación será abordado en los capítulos I.3 y en V.1).

Situación de análisis

Analice la secuencia de símbolos que se muestra el listado siguiente

```
class Persona
{
      string nombre;
      string apellidos;
      int añoNacimiento;

      int Edad(int unAño)
      {
            int tempEdad;
            tempEdad = unAño - añoNacimiento;
            return tempEdad;
      }

      string ToString()
      {
            return nombre + " " + apellidos;
      }
}
```

Pregunta: ¿Qué diferencias encuentra entre este listado y la variante de implementación de I.2.2.3?

```
      int tempEdad;
      tempEdad = unAño - añoNacimiento;
      return tempEdad;
```

en lugar de

```
      return unAño - añoNacimiento;
```

Este es un caso novedoso de declaración de variable porque se ha realizado una definición dentro del bloque de código del método. Nótese que en este caso no hay ningún problema pues lo que se ha hecho es descomponer la acción única de retornar el cálculo de la edad

```
     return unAño - añoNacimiento;
```

en tres acciones que incluyen:

La declaracion de la variable `tempEdad`

```
      int tempEdad;
```

La asignación del cálculo de la edad a la variable definida con anterioridad

```
      tempEdad = unAño – añoNacimiento;
```

Finalmente se retorna el valor que contiene la variable `tempEdad`

```
      return tempEdad;
```

Situación de análisis

Analice la secuencia de símbolos que se muestra el siguiente listado

```
class Persona
{
      string nombre;
      string apellidos;
      int añoNacimiento;
```

```
        int Edad(int unAño)
        {
              return unAño - añoNacimiento;
        }

        string ToString()
        {
              string tempString = nombre + " " + apellidos;
              return tempString;
        }
}
```

Pregunta: ¿Qué diferencias encuentra entre este listado y la variante de implementación de I.2.2.3?

```
string tempString = nombre + " " + apellidos;
return tempString;
```

en lugar de

```
return nombre + " " + apellidos;
```

De forma similar a la situación de análisis anterior, se ha descompuesto la acción única de retornar la cadena que concatena las variables `nombre` y `apellidos`

```
return nombre + " " + apellidos;
```

ahora en dos acciones que incluyen:

La declaración de la variable `tempString y` al mismo tiempo la asignación de la unión de las variables `nombre` y `apellidos`

```
string tempString = nombre + " " + apellidos;
```

Finalmente se retorna el valor que contiene la variable `tempString`

```
return tempString;
```

En determinados casos, C# asigna valores por defecto a las variables. Uno de estos casos es cuando se definen variables que representan campos. En la tabla siguiente se enumeran los valores que se asignan por defecto a este tipo de variable.

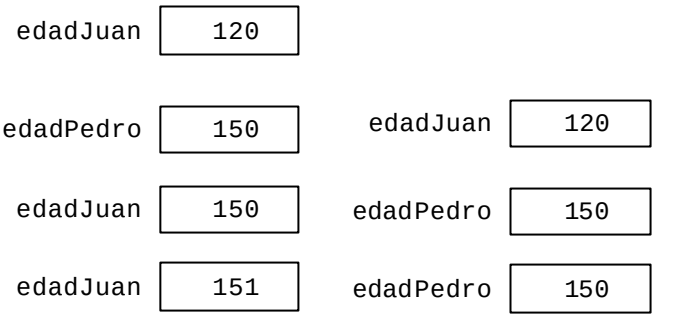
Tipo de variable	Valor por defecto
sbyte	0
byte	0
short	0
ushort	0
int	0
uint	0
long	0
ulong	0
float	0.0
double	0.0
decimal	0.0
bool	**false**
char	carácter Unicode con valor 0
string	**null**

Exceptuando los tipos de dato básicos, estructuras (Parte II) y tipos enumerados, el resto de los tipos utiliza semántica de copia por referencia, pues las variables de estos tipos apuntan a una zona de memoria (*Heap*) donde realmente se encuentra la direccion donde está ubicado el valor. Sin embargo, los tipos mencionados con anterioridad utilizan semántica de copia por valor pues el contenido de la variable es

realmente el valor asignado. Hay que destacar dentro de los tipos de dato básicos el caso de **`string`** que es un tipo que utiliza semántica por referencia; no obstante, en la asignación y comparación se comporta como un tipo con semántica por valor.

Acerca de los tipos de variables que utilizan semántica por referencia se dedican posteriores secciones del presente capítulo (I.2.6). Para simular lo que sucede realmente en memoria cuando se trabaja con variables que utilizan semántica de copia por valor está dedicada la descripción de forma grafica de lo que sucede con las siguientes variables:

```
{
    uint edadJuan = 120;
    uint edadPedro = 150;
    edadJuan = edadPedro;
    edadJuan++;
}
```

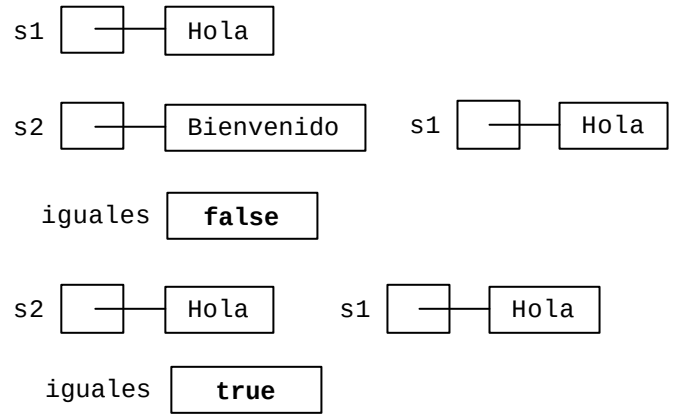


Seguidamente se comenta lo sucedido en cada una de las instrucciones:

- En las dos primeras instrucciones se les asignan a las variables `edadJuan` y `edadPedro` los respectivos valores `120` y `150`.
- En al tercera instrucción se le asigna a la variable `edadJuan` una copia del valor almacenado (contenido) en la variable `edadPedro`, es decir en este momento existen las dos variables con el mismo valor pero son independientes entre ellas.
- Finalmente la instrucción `edadJuan++;` solamente afecta el contenido de la variable `edadJuan`.

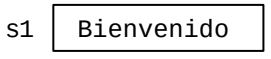
La representación en memoria de variables de tipo **`string`** se ejemplifica a continuación:

```
{
    string s1 = "Hola";
    string s2 = "Bienvenido";
    bool iguales = s1 == s2;
    s2 = s1;
    bool iguales = s1 == s2;
}
```



Como el tipo de dato **`string`** apenas utiliza la semántica por referencia solo para la representación interna, en lo adelante se usará para representar en memoria las variables de este tipo una notación semejante a la utilizada en el resto de los tipos básicos como se muestra a continuación:

```
string s1 = "Bienvenido";
```



Además de los elementos presentados con anterioridad, el operador de asignación tiene otra serie de funcionalidades como se muestra a continuación:

- En C#, el operador de asignación ("="), 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 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 en la edición de nuestros códigos a la hora de realizar asignaciones tan comunes como:

```
peso = peso + 10;     // Sin usar asignación compuesta
peso += 15;           // Usando asignación compuesta
```

  Las dos líneas anteriores son equivalentes, pues el operador compuesto += lo que hace es asignar a su primer operando el valor que tenía más el valor de su 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: +=, -=, *=, /=, %=, &=, |=, ^=, <<= y >>=. Nótese que no hay versiones compuestas para los operadores binarios && y ||.
- 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:

```
peso = peso + 1;  // Sin asignación compuesta ni incremento
peso += 1;        // Usando asignación compuesta
peso++;           // Usando incremento
```

  Si el operador ++ se coloca tras el nombre de la variable (como en el ejemplo) devuelve el valor de la variable antes de incrementarla, mientras que si se coloca antes, devuelve el valor de ésta tras 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.

I.2.4.2 Prioridad de los operadores. Arbol de evaluación

El resultado de una expresión depende del orden en que se ejecutan las operaciones, en el siguiente ejemplo se comprobará la importancia de este orden entre las operaciones.

Situación de análisis

Si no existiese un orden de prioridad entre los operadores, la siguiente expresión:

```
3 + 4 * 2;
```

pudiese arrojar dos resultados diferentes:

11 si se efectúa primero la multiplicación y después la suma y 14 en caso contrario.

Con el objetivo de que el resultado de cada expresión sea claro e inequívoco es necesario definir reglas que definan el orden en que se ejecutan las expresiones en C# (y en cualquier lenguaje).

Existen dos tipos de reglas para determinar este orden de evaluación: las reglas de precedencia y las de asociatividad.

Para la evaluación de las expresiones en C# se aplican las siguientes reglas cuando existe más de un operador:

- Si en la expresión todos los operadores tienen la misma prioridad la evaluación de estos se lleva a cabo de izquierda a derecha en unos y de derecha a izquierda en otros.
- Cuando existen operadores de diferente prioridad se evalúan primero los que tengan mayor prioridad, a continuación los de la prioridad siguiente y así sucesivamente.

- Las dos reglas anteriores quedan sin efecto cuando se incluyen paréntesis. En este caso la parte de la expresión encerrada entre paréntesis se evalúa primero siguiendo a su vez estas tres reglas.

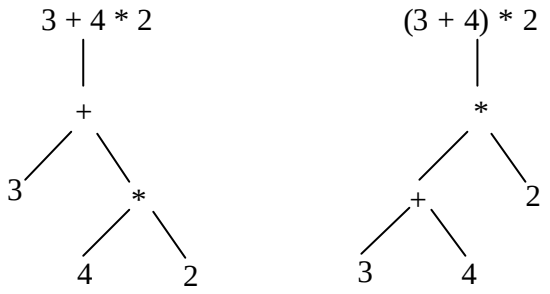
Seguidamente se muestra en una tabla la precedencia (ordenados de mayor a menor) y asociatividad de los operadores de C#.

Precedencia	Asociatividad
`() [] .`	izquierda a derecha
`++ -- ! + (unuario) – (unario)`	derecha a izquierda
`* / %`	izquierda a derecha
`+ –`	izquierda a derecha
`< <= > >=`	izquierda a derecha
`== !=`	izquierda a derecha
`&& &`	izquierda a derecha
`|| |`	izquierda a derecha
`?:`	derecha a izquierda
`= += -= *= /= %= &= |=`	derecha a izquierda
`,`	izquierda a derecha

Pregunta: ¿Cuál será entonces el resultado de la evaluación de la expresión de la situación de análisis presentada en el inicio de la sección (`3 + 4 * 2`)?

El resultado es 11 porque el operador * tiene mayor prioridad o precedencia que el operador +, por lo que en la expresión se efectúa primero la multiplicación y luego la suma.

Las expresiones se pueden representar de forma gráfica a través de los árboles de evaluación, informalmente se define un *árbol de evaluación* como una estructura jerárquica que relaciona un conjunto de *nodos.* Existe un nodo especial que se denomina nodo raíz donde se coloca la expresión, en los nodos intermedios se colocan los operadores que se aplican sobre sus hijos (note que siempre los nodos intermedios tienen hijos) y nodos finales u hojas donde aparecen los operandos. En la figura siguiente se muestran los respectivos árboles de evaluación de la expresión analizada y una variante de la misma forzando primero la suma a través del uso de paréntesis.



La evaluación de la expresión se realiza de abajo hacia arriba comenzando por los nodos intermedios que estén mas abajo hasta llegar al nodo que es hijo del nodo raíz. Por ejemplo, en el caso del primer árbol se evalúa primero el nodo "*" (4 * 2) y luego el nodo "+" (3 + 8).

I.2.5 Métodos

Un *método* es un conjunto de acciones que se agrupan a través de un determinado nombre simbólico de tal manera que es posible ejecutarlas en cualquier momento sin tenerlas que volver a escribir 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.

Los métodos son bloques de instrucciones que pueden devolver algún valor cuando se ejecutan. Invocar a un método hace que las instrucciones del mismo se ejecuten. En las instrucciones es posible acceder con total libertad a la información almacenada en los campos de la clase a la cual pertenece el método. A continuación se presenta de forma general la sintaxis de declaración de los métodos:

```
< Tipo devuelto > < Nombre simbólico del método > ([<Parámetros>])
{
      < Cuerpo del método >
}
```

A su vez todo método puede devolver un objeto como resultado de haber ejecutado las `<instrucciones>`. En tal caso el tipo del objeto devuelto tiene que coincidir con el especificado en `<TipoDevuelto>`, y el método tiene que terminar con la cláusula **`return`** `<objeto>;` En caso que se desee definir un método que no devuelva objeto alguno se omite la cláusula **`return`** y se especifica **`void`** como `<TipoDevuelto>`.

Opcionalmente todo método puede recibir en cada llamada una lista de parámetros a los que podrá acceder en la ejecución de las `<instrucciones>` del mismo. En `<Parámetros>` se indican los tipos y nombres de estos parámetros y es mediante estos nombres con los que se deberá referirse a ellos en el cuerpo del método.

Aunque los objetos que puede recibir el método como parámetro pueden ser diferentes cada vez que se solicite su ejecución (llamado), siempre han de ser de los mismos tipos y seguir el orden establecido en `<Parámetros>`.

La sintaxis usada para llamar a los métodos de un objeto es la misma que se utiliza para acceder a sus campos, solo que además tras el nombre del método que se desea invocar habrá que indicar entre paréntesis cuales son los valores que se desea dar a los parámetros. O sea se escribe de la siguiente forma:

```
<objeto>.<nombre del método>([<valor de los parámetros>])
```

`<objeto>` no tiene que ser explícitamente el nombre de una variable, es cualquier expresión que retorne como resultado un objeto.

Se debe recordar que en el caso de estudio I.1.8, el Estudiante juan puede responder a los mensajes para que informe su promedio de la siguiente forma:

```
juan.Promedio()
```

A continuación, se muestra una aproximación de la definición del método Promedio para la clase Estudiante.

```
class Estudiante;
{
      ...

      int Promedio()
      {
      ...
      }

      ...
}
```

Es el momento propicio para citar un par de notas que aparecen en [10], “los términos método y función suelen usarse de forma indistinta, pero hay una diferencia. Un método es una función contenida en una clase. Una función suele ser un grupo de instrucciones que no está contenido en una clase y que suele estar en un lenguaje, como C o C++. Como en C# no se puede añadir código fuera de una clase, usted nunca tendrá una función. Todos los métodos se encierran en una clase. Un método no puede existir fuera de una clase”.

I.2.5.1 Tipo devuelto

Pregunta: ¿Por qué el método `Edad` comienza con **`int`**?

La responsabilidad de este método es devolver la edad de una instancia de `Persona` y este valor devuelto siempre va a ser un valor que pertenece al dominio de valores del conjunto de los números enteros y una de las formas que existe en C# de representar enteros es a través del tipo de dato básico **`int`**.

Pregunta: ¿Siempre los métodos tienen que retornar un valor?

La respuesta en este caso es NO, tómese como ejemplo el método `Arranca()` de la clase `Cronómetro`. Este es un ejemplo de método que no tiene que retornar ningún valor, su responsabilidad es apenas echar a andar una instancia de `Cronómetro` y modificar su estado interno.

Para la definición de los métodos que no retornan ningún valor es utilizada en C# la palabra reservada **`void`,** como se muestra seguidamente a través del ejemplo de la clase `Cronómetro`.

```
class Cronómetro;
{
      ...

      void Arranca()
      {
            ...
      }

      ...
}
```

La ejecución de las acciones 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, por lo tanto se pueden escribir métodos que no devuelvan ningún valor.

I.2.5.2 Parámetros de los métodos

Otro elemento de interés en el método `Edad` de la clase `Persona` es la combinación de símbolos que aparece entre paréntesis, "`int unAño`".

Pregunta: ¿Cuáles son los datos o valores que necesita el método `Edad()` para cumplir con su responsabilidad de retornar el valor de la edad (con relación a un año determinado) de cualquier `Persona`?

En primer lugar necesita saber el año de nacimiento, hecho resuelto a través de la responsabilidad `añoNacimiento`. En segundo lugar necesita conocer con respecto a que año va a calcularse dicha edad y esto es desconocido en principio, además puede ser que en un momento quisiera saber la edad de una persona en el año actual o hace 5 años. Es por ello que aseguramos que le está faltando información al método `Edad` para cumplir con la respectiva responsabilidad asignada. La referida información que está faltando es la que se completa a través del *parámetro* `int unAño`.

Los parámetros son definidos entre paréntesis a continuación del nombre de la clase. Un método puede contener varios parámetros (lista de parámetros) o ninguno. En caso de contener una lista de parámetros éstos van antecedidos por su tipo y separados por coma. En caso contrario la lista de parámetros estaría vacía. Véase en la implementación de `Cronómetro` la definición del método `Arranca`.

Al conjunto formado por el nombre de un método y el número y tipo de sus parámetros se le conoce como *signatura* o *firma* del método. La signatura de un método es lo que verdaderamente lo identifica.

I.2.5.3 Cuerpo del método

Se le denomina instrucción a toda acción que se pueda realizar en el cuerpo de un método, como definir variables, llamar a métodos, asignaciones y muchas cosas más como veremos a lo largo de este libro. Las instrucciones se agrupan formando *bloques de instrucciones* (bloques de codigo), que son secuencias de instrucciones encerradas entre llaves que se ejecutan una tras otra.

El cuerpo de un método es el bloque de instrucciones que compone el código del método, por ejemplo

```
return unAño - añoNacimiento;
```

al tomar como referencia el método `Edad` de la clase `Persona`.

Nótese en el código anterior la existencia de la palabra reservada return para especificar precisamente el valor que va a retornar el método. Si se define un método que retorna un valor, en C# es responsabilidad del programador garantizar que durante el desarrollo de las acciones (instrucciones) del mismo se retorne siempre un valor, en caso contrario el compilador reportará un error.

En caso de que el método no tenga que retornar ningún valor, simplemente no hay que usar el return dentro del cuerpo.

I.2.5.4 Sobrecarga de métodos

Pregunta: ¿Es posible ofrecer otra variante para la responsabilidad que determina la edad en la clase `Persona` a través del mismo identificador `Edad`?

El referido método podría calcular la edad de una instancia de `Persona` a través del valor retornado por el reloj interno de la máquina (el año en este caso), si existiese alguna forma de obtener el mismo. Felizmente en la `BCL` existe una clase que dispone de esa responsabilidad (`System.DateTime.Now.Year`).

Lo interesante de este planteamiento es que ahora la propuesta del método `Edad` dispone de toda la información para dar un resultado ya que el año a comparar sería obtenido de la referida forma desde el reloj interno de la máquina.

Pregunta: ¿Esto implica que no se necesita el parámetro?

Por supuesto que no, tómese que por otra parte se quiere brindar las dos facilidades, una que calcule la edad a partir de un año dado y otra que la calcule a partir del año actual adquirido a través del reloj de la máquina. Evidentemente esta propuesta trae o conlleva a interrogantes interesantes, pues como se ha decidido brindar las dos facilidades se tienen que definir dos métodos para acceder a estas de forma independiente, puesto que sus propuestas de solución difieren notablemente.

Pregunta: ¿Qué nombres llevarán estos métodos?

Lo que supuestamente es una respuesta evidente, conlleva a otra interrogante aún más interesante. Como ambos métodos están definidos con el objetivo de calcular la edad de una instancia de `Persona` el nombre más lógico con el cual se debe hacer referencia a cada uno de ellos debe ser `Edad`, pues otro nombre se contradice con lo que realmente hace o con el objetivo para lo cual fue diseñado. Luego en la clase `Persona` existirían dos métodos con igual identificador, tipo devuelto y solo con la diferencia de que uno necesita de un parámetro para establecer con respecto a que año se calcularía la edad, mientras que el otro no necesita del parámetro porque la calcula respecto al año actual tomado del reloj de la máquina.

Pregunta: ¿Puede contener una clase más de un método con igual identificador? En caso positivo, ¿cómo el compilador es capaz de diferenciarlos?

La respuesta a la primera pregunta es afirmativa, C# permite que en una misma clase se definan varios métodos con el mismo nombre siempre y cuando tomen diferentes número o tipo de parámetros, o que en caso extremo si tienen igual número y tipo de parámetros se diferencien en el orden de los mismos.

Situación de análisis

Analice la siguiente definición de clase.

```
class A
{
      ...

      int f(int x, int y){...}

      int f(string x, int y){...}

      int f(int x, int y, int z){...}
```

```
        int f(int y, string x){...}

        int f(int a, int b){...}
}
```

Pregunta: ¿Qué reacción tiene el lenguaje ante las definiciones anteriores?

El ejemplo anterior consta de 5 definiciones del método f. Las cuatro primeras son aceptadas por el lenguaje, ya que difieren en el tipo, la cantidad u orden de los parámetros. El último caso no es correcto pues solo se diferencia del primero en los identificadores de los parámetros. Este produce un error semántico puesto que existe un método previamente definido que tiene dos parámetros enteros. Es decir, el nombre de los parámetros, no establece diferencia alguna para la signatura de distintos métodos.

A la posibilidad de disponer de varios métodos con el mismo nombre, pero con diferente lista de parámetros es lo que se conoce como *sobrecarga de métodos* y es posible ya que cuando se les invoque el compilador podrá determinar a cual llamar a partir de los parámetros pasados en la llamada.

Sin embargo, lo que no se permite es definir varios métodos que solamente se diferencien en su valor de retorno, puesto que la forma de invocar a los métodos a través de instancias de la clase es

```
<objeto>.<nombre del método>([<valores de los parámetros>])
```

es apreciable que el tipo devuelto no tiene que especificarse en el llamado por lo que la existencia de varios métodos, que solo se diferencien en este valor se retorno se consideraría un error, ya que no se podría diferenciar a que método en concreto se hace referencia en la llamada.

Situación de análisis

Analice la siguiente definición de clase.

```
class A
{
        ...

        int f(int x, int y){...}

        string f(int x, int y){...}
}
```

Pregunta: Si esto fuera permitido y se tiene entonces que `a1` es una instancia de `A`. ¿Qué sucedería con una instrucción de la forma `a1.f(2, 4)`?

Evidentemente, no se sabría a cual de las dos variantes de método llamar pues como se había referido con anterioridad el tipo devuelto no establece diferencias en la llamada a los métodos.

Ejemplo: Refine la clase `Persona` implementando los dos métodos `Edad` propuestos con anterioridad.

La solución a este ejemplo se muestra en el siguiente listado.

```
class Persona
{
        ...

        uint añoNacimiento;

        int Edad(int unAño) {...}

        int Edad()
        {
                return System.DateTime.Now.Year - añoNacimiento;
        }
}
```

Como se ha visto es posible definir en una misma clase varios métodos con idéntico nombre, siempre y cuando tengan distintos parámetros, cuando esto ocurre se dice que el método que tiene ese nombre está *sobrecargado*. La sobrecarga de métodos se establece entre métodos con igual identificador y diferente signatura.

La sobrecarga de métodos permite entonces realizar las siguientes operaciones sobre una instancia `juan` de `Persona` como se muestra seguidamente.

```
{
      ...
      juan.Edad(2003);

      ...

      juan.Edad();
}
```

I.2.6 Creación de objetos. Manejo de las instancias

Hasta el momento se han abordado algunos aspectos concernientes a cómo se deben implementar las clases en C#, cómo expresar sus campos y métodos y cómo acceder a estos a través de instancia de las respectivas clases.

Aún cuando en C# todos los tipos de dato son clases, en el presente texto se denominarán objetos a las instancias de las clases que excluyen los tipos de dato básicos del lenguaje.

Recuerde entonces que en el caso de los tipos de dato básicos sus valores pueden asociarse a la definición de variables y luego a través del operador de asignación (=), éstas cambiarían su estado.

Pregunta: ¿Es factible seguir esta misma idea para manejar objetos?

Para manipular objetos o instancias de tipos de dato que no sean básicos (de otras clases para ser más precisos) también se utilizan variables y éstas utilizan semántica por referencia, solo que con una diferencia con respecto a los **`strings`**, los objetos tienen que ser creados (tómese por crear un objeto el hecho de reservar memoria para almacenar los valores de todos sus campos) explícitamente a través del operador **`new`** o al menos contener la referencia de otro objeto creado con anterioridad. En caso contrario contendrán una referencia a **`null`**, lo que semánticamente significa que no está haciendo referencia a ningún objeto.

Pregunta: ¿Cómo funciona el operador **`new`**? ¿Cuál es su sintaxis y semántica?

Sintaxis: **`new`** `<nombreTipo>([<valores de los parámetros>])`

Este operador crea un objeto de `<nombreTipo>` pasándole a su método *constructor* los parámetros indicados en `<parámetros>`. Una vez que se ha creado el objeto deseado lo más lógico es almacenar la referencia devuelta por **`new`** en una variable del tipo apropiado para el objeto creado, de forma tal que se facilite su posterior manejo. El siguiente ejemplo muestra como crear objetos de tipo `Persona` y como almacenarlos en variables.

Ejemplo: Definir algunas instancias de la clase `Persona`.

```
{
      Persona maría;
      maría = new Persona();
      Persona pedro, ana;
      Persona juan = new Persona();

}
```

Pregunta ¿Qué significa entonces la instrucción

```
maría = new Persona();
```
?

Esta es la forma de crear una instancia de `Persona` y asignarla a la variable `maría`, después de ello `maría` contiene una referencia a la zona memoria donde "realmente" existe el objeto creado.

I.2.6.1 Constructores

En la sección anterior se ha hablado sobre los constructores y se ha dicho a grandes rasgos que son métodos.

Pregunta: ¿Qué son realmente los constructores? ¿Cuál es su función? ¿Cuál es su signatura? ¿Cómo funcionan?

Los constructores de una clase son métodos especiales que se definen como componentes (responsabilidades) de ésta y que contienen código a ejecutar cada vez que se cree un objeto de ese tipo. Éste código suele usarse para la inicialización de los atributos del objeto a crear, sobre todo cuando el valor de éstos no es constante o incluye acciones más allá de una asignación de valor. La sintaxis básica de definición de constructores consiste en definirlos como cualquier otro método pero dándoles el mismo nombre que la clase a la que pertenecen y no indicando el tipo de valor de retorno.

Como **`new`** siempre devuelve una referencia a la dirección de memoria donde se crea el objeto y los constructores solo pueden utilizarse como operandos de **`new`** , no tiene sentido que un constructor devuelva objetos por lo que incluir en su definición un campo `<TipoDevuelto>` se considera erróneo (incluyendo **`void`**).

La sintaxis apropiada para definir constructores es:

```
<nombreTipo>([<parámetros>])
{
      <código>
}
```

Pregunta: ¿Cómo es posible crear instancias de la clase `Persona` si no se han definido explícitamente algún constructor para la misma? ¿Es obligatorio definir constructores?

No es obligatorio definir constructores y en el caso en que no se definan (como en el ejemplo anterior de la clase `Persona`) C# brindará uno sin parámetros y de cuerpo vacío (constructor por defecto). Dicho constructor solo se ocupa de asignarle a todos los campos sus valores por defecto.

Hay que tener en cuenta que este constructor será incluido por el compilador solamente si no se ha definido ningún otro de forma explícita. Luego, si se tiene una clase para la cual se ha definido al menos un constructor con parámetros y ninguno sin parámetros, cualquier intento de crear objetos utilizando el constructor por defecto será reportado por C# como error.

Por el momento para la clase `Persona`, se tienen que crear las instancias y más tarde sustituir los valores por defecto que le han sido asignado a sus atributos por los valores precisos del objeto en cuestión como se muestra a continuación:

```
{
     Persona maría = new Persona();
     maría.nombre = "María";
     maría.apellidos = "González";
     maría.añoNacimiento = 1980;
}
```

Pregunta: ¿Sería posible unificar el proceso de creación de la instancia y la inicialización de sus campos?

Claro que si, y la solución es sencilla. Basta con definir explícitamente un constructor que reciba como parámetros los valores que se utilizarán para la inicialización de los campos del objeto y que contenga en su cuerpo dicha inicialización de los campos.

Ejemplo: Definir un constructor que permita inicializar las instancias de la clase `Persona` con valores específicos para sus campos.

```
class Persona
{
      string nombre;
      string apellidos;
      uint añoNacimiento;

      // definición de un constructor
      Persona(string unNombre, string unAp, int unAñoNacimiento)
      {
            nombre = unNombre;
            apellidos = unAp;
            añoNacimiento = unAñoNacimiento;
      }

      int Edad(int unAño) …
}
```

La disponibilidad de este constructor permite crear objetos Persona como se muestra a continuación:

```
Persona maría = new Persona("María", "González", 1980);
```

Pregunta: Retomando la implementación anterior, ¿Qué sucede si los parámetros del constructor se denominan igual que los campos? ¿Dentro del cuerpo del constructor a quien se haría referencia al campo o al parámetro?

La respuesta a estas interrogantes se pueden obtener a partir de los conceptos de *alcance* (*ámbito*) y *tiempo de vida* de las variables.

Tiempo de vida de las variables: Es el tiempo durante el cual se puede hacer referencia a la variable y va desde su creación hasta que dejan de existir. Para el caso de los campos de los objetos, estos existen durante toda la vida de los objetos. Pero en el caso de las variables locales (ejemplo, los parámetros), su tiempo de vida es mas limitado pues dejan de existir después que finalice el bloque de instrucciones al que pertenece.

Alcance o *ámbito*: Este concepto esta relacionado con la visibilidad de las variables y especifica desde que código es posible hacer referencia a la variable.

Lo que supuestamente podría dar un error por una posible redefinición de la variable `nombre` (por ejemplo) no sucede, puesto que la definición del parámetro se realiza en un bloque más interno que la del campo. El parámetro, durante su tiempo de vida, oculta al campo.

Para solucionar esta problemática, C# ofrece la palabra reservada **`this`**, que permite referenciar al objeto en curso y uno de sus usos es diferenciar campos y parámetros (entre otras cosas) con iguales identificadores, en el ámbito que sea necesario como se muestra en el siguiente listado.

```
class Persona
{
      string nombre;
      string apellidos;
      uint añoNacimiento;

      Persona(string nombre, string apellidos, int añoNacimiento)
      {
            this.nombre = nombre;
            this.apellidos = apellidos;
            this.añoNacimiento = añoNacimiento;
      }
      int Edad(int unAño) …
}
```

La palabra reservada **`this`** identifica al objeto en curso o actual, o sea, al objeto para el cual se está ejecutando el código del método en cuestión. Esto permite que a través de **`this`** se haga referencia a las responsabilidades de este objeto.

Para ejemplificar que sucede en memoria cuando se trabaja con objetos, se desarrolla la situación de analisis siguiente auxiliándose de una representación grafica:

Situación de análisis

Analice el segmento de código que se muestra a continuación:

```
{
      Persona juan = new Persona("Juan", "Ferrer", 1980);

      Persona pedro = juan;

      pedro.nombre = “Pedro”;

      Console.WriteLine(pedro.nombre);

      Console.WriteLine(juan.nombre);

}
```

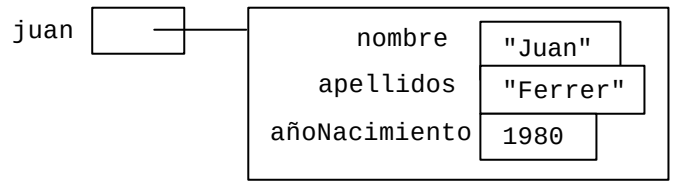
Pregunta: ¿Qué se imprimen al ejecutar el código del listado con anterioridad?

Se imprimen los siguientes mensajes:

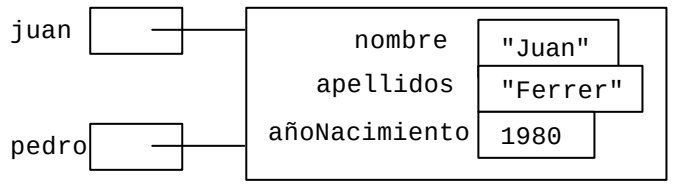
```
Pedro
Pedro
```

Esto se justifica por la semántica por referencia que utilizan los objetos. Es decir, la variable `pedro` y la variable `juan` hacen referencia al mismo objeto por lo que cualquier cambio en una de ellas afecta a la otra. Ver la representación de este suceso en la figura siguiente.

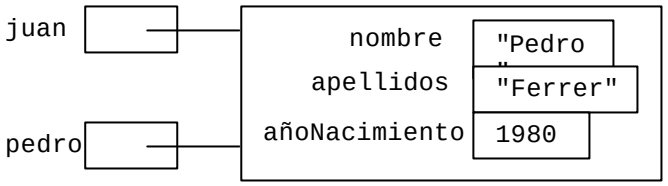
a) `Persona juan = new Persona("Juan", "Ferrer", 1980);`



b) `Persona pedro = juan;`



c) `pedro.nombre = “Pedro”;`



Seguidamente se comenta lo sucedido en cada una de las instrucciones:

- En la primera instrucción se reserva un espacio en memoria para almacenar los valores de los campos de un objeto de tipo `Persona`, seguidamente se inicializan dichos campos con los valores pasados a su constructor y la referencia devuelta por el operador **new** se almacena en la variable `juan`.
- En la segunda instrucción se le asigna a la variable `pedro` de tipo `Persona` la referencia al objeto creado con anterioridad.
- En la tercera se modifica el campo nombre del objeto referenciado por la variable `pedro`, que es el mismo objeto referenciado por la variable `juan`.
- En las dos instrucciones siguientes se imprimen los campos nombre de las variables `pedro` y `juan` respectivamente; `pedro` y `juan` son variables que contienen referencias al mismo objeto, por lo tanto en ambos casos se imprime el mismo valor.

I.2.7 Estilo de código

Situación de análisis

Analice la secuencia de símbolos que se muestra a continuación

```
public class P { string n; string a; int aN;/* identificacion  de la clase */


public int E(int uA)
      {

      return uA–aN
// cálculo de la edad
;
}

      public string TS()
{
            return n+“ ”+a;
      }
      }
```

Esta clase es equivalente a la clase `Persona` desarrollada en la sección I.2.2.3.

Pregunta: ¿Por puede ser complicado llegar a la conclusión anterior?

En ello han incidido varios elementos entre los que se pueden relacionar:

- No se han usado identificadores nemotécnicos.
- Los comentarios no cumplen con su función esclarecedora.
- Los componentes de la clase no se han organizado correctamente, etc.

Se define como estilo de código a un conjunto de reglas que permitan dar un buen nivel de homogeneidad al código fuente y su objetivo es facilitar la lectura de dicho código, principalmente a otros programadores que no participaron en la elaboración del mismo.

Aunque cada programador puede definir su propio estilo de código, una buena práctica es seguir el estilo utilizado por los diseñadores del lenguaje pues de seguir esta práctica será mucho más fácil analizar códigos de otros y a su vez que otros programadores analicen y comprendan nuestros códigos.

Algunas recomendaciones para el estilo de código en C#.

- Evitar en lo posible líneas de longitud superior a 80 caracteres.
- Indentar los bloques de código.
- Las llaves de apertura ("{") se colocarán siempre en una sola línea, inmediatamente después de la línea de la instrucción que da pie a su uso y con la misma indentación de dicha instrucción y las de cierre ("}") se colocarán de igual forma en una sola línea coincidiendo con su respectiva llave de apertura.
- Los operadores se separarán siempre de los operandos por un carácter "espacio". La única excepción para esta regla es el operador de acceso a los componentes de objetos y clases (".") que no será separado.
- Utilizar identificadores nemotécnicos, es decir, utilizar nombres simbólicos adecuados para los identificadores lo suficientemente autoexplicativos por si mismos para dar una orientación de su uso o funcionalidad de manera tal que podamos hacer más claros y legibles nuestros códigos.
- Los identificadores para constantes se especificarán siempre en mayúsculas.
- Los identificadores de variables y parámetros se especificarán siempre comenzando en minúsculas, si se compone de más de un nombre, entonces el segundo comenzará con mayúscula.
- Los identificadores de clases, métodos y propiedades se especificarán siempre comenzando en mayúsculas, si se compone de más de un nombre, entonces el segundo también comenzará con mayúscula.
- Utilizar comentarios, pero éstos seguirán un formato general de fácil portabilidad y que no incluya líneas completas de caracteres repetidos. Los que se coloquen dentro de bloques de código deben aparecer en una línea independiente indentada de igual forma que el bloque de código que describen. Aún cuando los programadores pueden crear sus propios estilos para utilizar y escribir los comentarios, recomendamos tener en cuenta algunos que se muestran en [8] y que se referencian en I.2.2.5.

I.2.8 Secuenciación. Caso de estudio: clase `Temperatura`

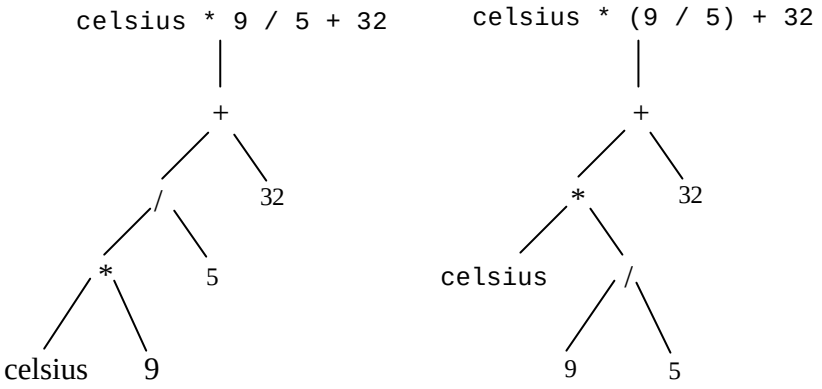
Ejemplo: Diseñe e implemente una clase `Temperatura` con responsabilidades para almacenar un valor de temperatura en grados Celsius (C) y obtener su correspondiente en grados Fahrenheit (F) (F = C * 9 / 5 + 32) (idea original en [8,9]).

Para comenzar es necesario determinar las responsabilidades y clasificarlas en variables y métodos. A partir del enunciado queda explícito que al menos se tiene que implementar una responsabilidad para almacenar un valor de temperatura en grados Celsius, lo cual constituye un dato o información por lo que se implementa a través de una variable; por otra parte se debe poder obtener la conversión de este valor a grados Fahrenheit donde evidentemente intervienen acciones u operaciones por lo que su representación es a través de un método (en la implementación de este método apenas interviene una expresión aritmética). El diseño e implementación de esta clase se muestran a continuación:

Temperatura
double celsius
Temperatura(**double** unaTempCelsius) **double** Fahrenheit()

```
class Temperatura
{
    double celsius;

    Temperatura(double unaTempCelsius)
    {
          celsius = unaTempCelsius;
    }

    double Fahrenheit()
    {
          return celsius * 9 / 5 + 32;
    }
}
```

Ejemplo: A través de su respectivo árbol de evaluación, evaluar la expresión que se presenta en el método `Fahrenheit` del listado anterior para `celsius` == 100 (`celsius * 9 / 5 + 32`). Para el mismo valor de la variable `celsius`, evaluar la expresión: `celsius * (9 / 5) + 32`.

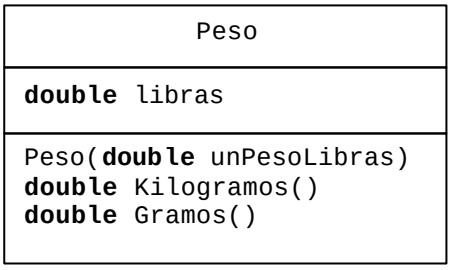


A pesar de que en principio son similares, la evaluación de estas expresiones produce resultados diferentes: 212 en el primer caso (`celsius * 9 / 5 + 32`) y 132 en el segundo. Esto esta dado porque en el segundo caso el primer término en avaluarse es `9 / 5` y este produce como resultado 1 porque al estar presentes dos valores enteros, el resultado de la operación también es un valor entero. Para solucionar esta problemática simplemente se tendría que expresar al menos uno de los valores que intervienen en la expresión como valor constante (`9.0 / 5` ó `9 / 5.0` ó `9.0 / 5.0`).

I.2.9 Secuenciación. Caso de estudio: clase `Peso`

Ejemplo: Diseñe e implemente una clase `Peso` con responsabilidades para almacenar un peso en libras y obtener su equivalente en kilogramos y gramos. (Una libra equivale a 0.453592 kilogramos) (idea original en [9]).

Analizando el enunciado anterior queda explícito que al menos tenemos que implementar una responsabilidad para almacenar el valor de un peso en libras, el cual también constituye un dato o información (de forma análoga a la temperatura en grados Celsius) por lo que se implementa a través de una variable; por otra parte se debe poder obtener la conversión de este valor a gramos y kilogramos donde evidentemente también intervienen acciones u operaciones por lo que estamos en presencia de dos métodos. En la implementación de estos métodos veremos como hecho significativo la reusabilidad de código puesto que si se dispone de una expresión para la conversión a kilogramos, se utiliza el valor convertido en kilogramos para la conversión a gramos. El diseño e implementación de esta clase se muestran en la figura y el listado siguientes:




```
class Peso
{
    double libras;

    Peso(double unPesoLibras)
    {
          libras = unPesoLibras;
    }

    double Kilogramos()
    {
          return libras * 0.453592;
    }

    double Gramos()
    {
          /* aquí se reutiliza código puesto que se llama al
             método Kilogramos() en vez de colocarse explícitamente la
             expresión para el cálculo de este
          */
          return Kilogramos() * 1000;
    }
}
```

Ejemplo: En el listado que se muestra a continuación, determinar qué elementos del estilo de código propuesto en I.2.7 han sido ignorados.

```
public class P
{
double Libras;
public P(double unPesoLibras)
{
Libras = unPesoLibras;
}
public double kilogramos()
{
return libras * 0.453592;
}
public double gramos()
{
return kilogramos()*1000;
}
}
```

Resulta evidente la poca legibilidad que presenta el código mostrado en el listado anterior y en esto incide determinantemente que se han ignorado los siguientes elementos del estilo de código propuesto en I.2.7:

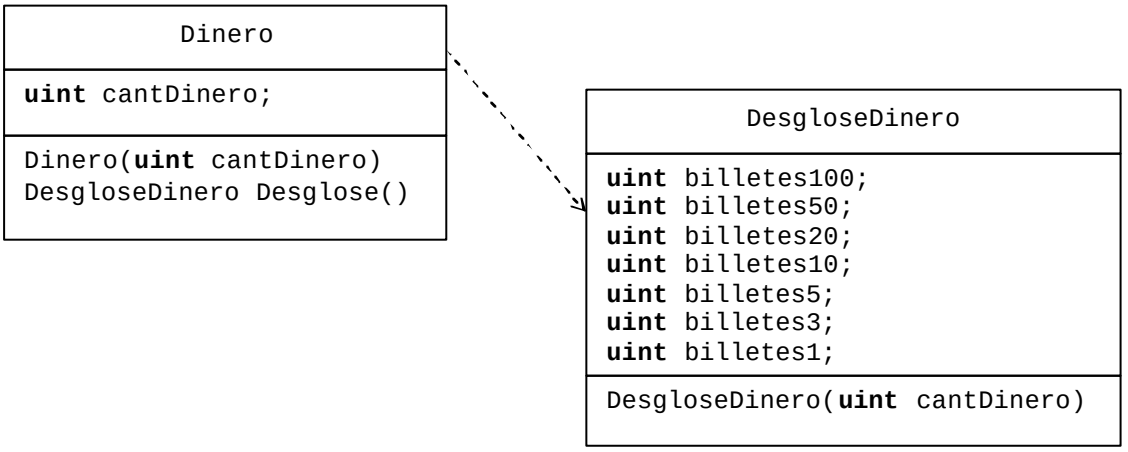
- El identificador de la clase (P) no es nemotécnico.

- No se han indentado los correspondientes bloques de código.
- El identificador de variable `Libras` comienza con mayúscula, mientras que los de los métodos `kilogramos` y `gramos` comienzan con minúscula.
- No se han utilizado comentarios.
- En la expresión `kilogramos()*1000`, no se han separado los operandos (`kilogramos()` y `1000`) del operador (`*`).

I.2.10 Secuenciación. Caso de estudio: clase `DesgloseDinero`

Ejemplo: Diseñe e implemente una clase (`Dinero`) que almacene una cantidad de dinero y permita obtener su desglose según las monedas y billetes habituales en Cuba (100, 50, 20, 10, 5, 3, 1). Se desea encontrar el número mínimo de billetes y monedas. No hay limitación en el número de billetes y monedas disponibles (idea original en [9]).

A partir del análisis del enunciado anterior es evidente que al menos se tiene que definir una variable para almacenar la cantidad de dinero. También en este momento debe quedar claro la implementación del desglose a través de un método sin embargo existe una diferencia con los métodos implementados con anterioridad: el resultado de este método no es un valor de un tipo simple. Una posible solución sería definir una nueva clase `DesgloseDinero` que tenga como variables la propia magnitud a desglosar y para expresar el desglose en cada una de los respectivos billetes; esta clase permitiría establecer una relación de uso con la clase `Dinero`. Cobra un interés especial el algoritmo para realizar el desglose. El diseño e implementación de estas clases se muestran en la figura y el listado siguiente.




```
class DesgloseDinero
{
    uint billetes100;
    uint billetes50;
    uint billetes20;
    uint billetes10;
    uint billetes5;
    uint billetes3;
    uint billetes1;

    DesgloseDinero(uint unaCantDinero)
    {
          uint restoDinero = unaCantDinero;

          billetes100 = restoDinero / 100;
          restoDinero = restoDinero % 100;
```

```
            billetes50 = restoDinero / 50;
            restoDinero = restoDinero % 50;
            billetes20 = restoDinero / 20;
            restoDinero = restoDinero % 20;

            billetes10 = restoDinero / 10;
            restoDinero = restoDinero % 10;

            billetes5 = restoDinero / 5;
            restoDinero = restoDinero % 5;

            billetes3 = restoDinero / 3;

            billetes1 = restoDinero % 3;
        }
    }

    class Dinero
    {
        uint cantDinero;

        Dinero(uint unaCantDinero)
        {
            cantDinero = unaCantDinero;
        }

        DesgloseDinero Desglose()
        {
            return new DesgloseDinero(cantDinero);
        }
    }
```

En la implementación de los algoritmos, la mayoría de las ocasiones, no existe una única acción que a partir del estado inicial, genere un estado final. Por esta razón, es necesario *descomponer* el algoritmo en una secuencia ordenada de acciones e introducir una serie de *estados intermedios* que cubran el salto entre el estado inicial y el estado final, reduciendo la complejidad del algoritmo [9].

La mayoría de los algoritmos diseñados hasta el momento cuentan apenas con una única acción o instrucción por lo que no ha quedado explícita la esencia de la *secuenciación* que se basa en la introducción de estados intermedios como forma de llegar del estado inicial al final. En el caso del algoritmo para obtener el desglose de dinero tenemos un estado inicial que es la cantidad de dinero a desglosar y como estado final la descomposición en los diferentes billetes. Sin embargo, para llegar a esta descomposición una variante de algoritmo sería comenzar obteniendo la cantidad de billetes de a 100 (dividiendo el valor inicial por 100), luego para obtener la cantidad de billetes de 50 se precisa saber qué dinero ha quedado luego de realizada la operación anterior (resto de la división) lo que constituiría un valor intermedio (variable `restoDinero` del listado anterior); de forma semejante se obtendrían los billetes de 20, 10, etc.

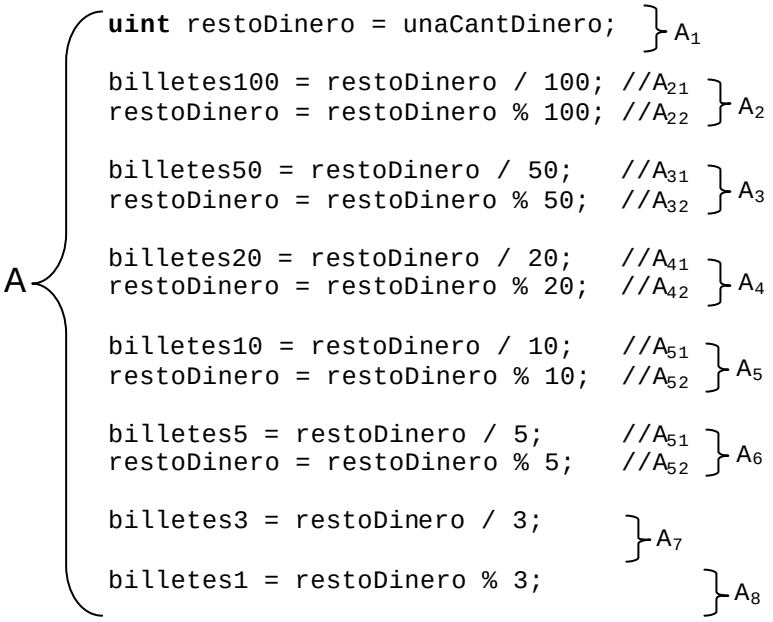
Hasta el momento se habían presentado constructores donde la inicialización de las variables se realiza asignando valores directamente. Sin embargo, en el constructor de la clase `DesgloseDinero` se aprecia un caso diferente pues para la inicialización de las respectivas variables se ha tenido que diseñar un algoritmo de *secuenciación* (*composición secuancial*) por lo que partiendo de un estado inicial (representado en el parámetro `cantDinero`) y atravesando por estados intermedios (representados en la variable `restoDinero`) se llega a un estado final (representado en los valores de las variables `billetes100`, `billetes50`, `billetes20`, etc).

A través de la *secuenciación* o *composición secuencial* es posible resolver un problema mediante un algoritmo, especificando las condiciones que se tienen que cumplir al inicio de la ejecución (estado inicial), y las condiciones que se deben cumplir al final de la ejecución (estado final) [9].

También en el constructor de la clase `DesgloseDinero` se tiene un ejemplo de método que representa una *acción* que permite consultar o modificar el valor de una o más variables. Una *acción* con duración finita que comienza en un instante inicial y finaliza en un instante final, produciendo un resultado bien definido que se refleja en los valores de variables.

Nótese que la descomposición secuencial permite ordenar secuencialmente un conjunto de acciones, tal y como se muestra en la figura siguiente, en la que la acción A, se descompone en primer lugar en una secuencia de acciones A_1, A_2,.. A_8 y posteriormente la acción A_2 se descompone en A_{21} y A_{22}, A_3 se

descompone en A_{31} y A_{32}, etc. Esto quiere decir que para alcanzar el estado final, primero se ejecuta A_1, luego A_{21} y A_{22} y así sucesivamente hasta llegar a A_8.



Otro elemento interesante en el listado propuesto como solución al ejemplo de la presente sección se presenta en el método `Desglose` de la clase `Dinero`, pues en lugar de retornar un valor de un tipo de dato simple (como se había realizado hasta el momento), se retorna una instancia de la clase `DesgloseDinero`:

```
return new DesgloseDinero(cantDinero).
```

Situación de análisis

Vale destacar que se ha realizado el anterior diseño de clases para tratar de coincidir "exactamente" con el enunciado del ejemplo introductoria de la sección, sin embargo simplemente con la clase `DesgloseDinero` casi se tiene una solución al referido ejercicio. Apenas faltaría designar una responsabilidad para representar la cantidad de dinero.

Pregunta: ¿Cómo es posible representar la responsabilidad para representar la cantidad de dinero?

Una variante para esta responsabilidad podría ser la definición de una variable `int cantDinero`, sin embargo otra variante podría ser a través de un método pues una vez que se tiene el desglose se puede obtener la cantidad de dinero que este representa como se muestra en el listado a continuación:

```
class DesgloseDinero
{
    ...

    uint CantDinero()
    {
          return billetes100 * 100 +
                 billetes50 * 50 +
```

```
                billetes20 * 20 +
                billetes10 * 10 +
                billetes5 * 5 +
                billetes3 * 3 +
                billetes1;
        }
}
```

Nota: Las variantes propuestas con anterioridad pueden pueden hacernos llegar a una importante conclusión: en el modelo OO se pueden encontrar disímiles soluciones para dominios de problemas similares; en esto tiene una incidencia muy significativa la ausencia de heurísticas formales para la determinación y clasificación de las responsabilidades. Teniendo en cuenta conclusión, entonces usted no tiene que preocuparse cuando su solución no coincida con la que se muestra en este texto: recuerde que solamente se desea orientar y ayudar a dar los primeros pasos en el fascinante mundo de la programación y particularmente en el paradigma OO por lo que se tratará de buscar soluciones didácticas.

También en las variantes anteriores se tiene la presencia de un elemento de suma importancia en programación que es la separación entre **el qué** y **el cómo**, o lo que es lo mismo la distinción entre *especificación* e *implementación*. Esto es, la distinción entre qué hace una determinada clase y cómo lo hace.

Al analizar los elementos expuestos en esta sección se aprecia la aplicación de algunas ideas expuestas en [9] y que se relacionan a continuación:

Un algoritmo especifica la secuencia de acciones que al aplicarse sobre un conjunto de informaciones produce el resultado deseado. Por tanto, el proceso de creación de un algoritmo consiste en *seleccionar* los conjuntos de informaciones y acciones, y a continuación decidir cómo *organizar* las acciones en el tiempo. En el caso de un proceso algorítmico, las informaciones, también denominadas variables, constituyen las magnitudes o indicadores que caracterizan tal proceso. La ejecución de las acciones puede provocar cambios en las informaciones.

Ejemplo: En el listado de la solución original, determinar algunos bloques de código, identificadores, palabras reservadas, comentarios, declaraciones de variables, constantes, tipos de dato básicos y expresiones.

Ejemplo de **bloque de código**

```
{
    this.cantDinero = cantDinero;
}
```

Identificadores: `DesgloseDinero`, `billetes100`, **`public`**, **`class`**, etc.

Palabras reservadas: **`public`**, **`class`**, **`uint`**, **`return`**, etc.

Comentarios: *no existen.*

Declaración de variables: **`uint`** `billetes50`, **`uint`** `billetes20`, **`uint`** `billetes10`, **`uint`** `restoDinero`, etc.

Declaración de parámetros: **`uint`** `unaCantDinero`.

Constantes: 100, `50`, 10, 20, etc.

Tipos de dato básicos: **`uint`**.

Expresiones: `restoDinero = cantDinero`, `billetes100 = restoDinero / 100`, etc.

Ejemplo: En el listado que se muestra a continuación, determinar los errores existentes y clasificarlos en sintácticos y semánticos.

```
class DesgloseDinero
{
     uint cantDinero;
     uint billetes100;
     uint billetes 50;
     uint #billetes20;

     DesgloseDinero(string cantDinero)
     {
           this.cantDinero = cantDinero;

           restoDinero = cantDinero;

           uint restoDinero = restoDinero % 100;

           billetes100 = cantDinero / 100;
}
```

El desarrollo de este ejemplo se presenta en la tabla siguiente:

Construcción	Tipo de error	Descripción
`uint billetes 50;`	sintáctico	se coloca un identificador seguido de un número en la definición de una variable, el compilador exigirá un punto y coma después del símbolo `billetes`; posible solución `billetes50`
`uint %billetes20;`	sintáctico	identificador no valido`%billetes20`, tiene que comenzar con letra o subrayado; posible solución `billetes20`
`this.cantDinero = cantDinero;`	semántico	`this.cantDinero` es de tipo entero, mientras que el parámetro `cantDinero` es una cadena de caracteres; esta conversión no se puede hacer de forma implícita; posible solución, declarar el parámetro también entero
`restoDinero = cantDinero;`	sintáctico	el identificador `restoDinero` no ha sido defunido previamente; posible solución `uint restoDinero`
`billetes100 = cantDinero / 100;`	semántico	`cantDinero` es una cadena de caracteres y no es posible aplicar el operador "`/`"entre una cadena y un valor entero; posible solución, declarar el parámetro `cantDinero` también entero
bloque de instrucciones del constructor `DesgloseDinero` (o el de la propia clase)	sintáctico	El bloque no ha sido cerrado, falta un paréntesis cerrado antes o después del que ya existe

I.2.11 Secuenciación. Caso de estudio: clase `Cronómetro`

Ejemplo: Diseñar e implementar una clase Cronómetro, en esta al menos deben existir responsabilidades para (idea original en [3]):

- Echar a andar.
- Parar.
- Retornar el tiempo acumulado en milisegundos.

Auxíliese de la clase Environment y su método TickCount que retorna el tiempo actual de la máquina en milisegundos.

Una variante de esta clase Cronómetro se muestra en el listado siguiente:

```
class Cronómetro
{
        int arrancóEn;
        int paróEn;

        void Arranca()
        {
              arrancóEn = Environment.TickCount;
        }

        void Para()
        {
              paróEn = Environment.TickCount;
        }

        public long Tiempo()
        {
              return paróEn - arrancóEn;
        }
  }
```

Vale destacar que esta implementación de la clase `Cronómetro` aún tiene que ser refinada, nótese que por ejemplo no se controla cuando el cronómetro está andando y cuando está parado; para este refinamiento se necesita del análisis de casos que se presenta en II.2.

I.2.12 Ejercicios

1. En el listado que se muestra a continuación determine los errores existentes y clasifíquelos en sintácticos y semánticos.

```
class Peso()
{
   double libras;
   Peso(real unPesoLibras)
   {
         libras == unPesoLibras;
   }
   double gramos()
   {
         const string librasGramos =  0.453592
         return libras * librasGramos;
   }
   double Kilogramos()
   {
         return Gramos() / 1000;
   }
}
```

2. Determine cuales de las siguientes cadenas de símbolos pueden constituir expresiones aritméticas sintácticamente válidas en C#:

a) A + (B * 2)

b) +(A + (A + A))

c) – X

d) A* - B

e) PiSobre2

f) (A/ B) – (C /D * E)

3. Desarrolle los árboles de evaluación y evalúe las siguientes expresiones aritméticas:

a) (A / B) – (C / (D * E))

b) A / B – (C / (D * E))

c) A / B – (C / D * E)

d) (A + B*C) / ((D * E) – (A * C))

e) A + (B*C) / (D * E) – (A * C)

4. Seleccione algunos de los listados del presente capítulo y determine bloques de código, identificadores, palabras reservadas, comentarios, declaraciones de variables, constantes, tipos de dato básicos y expresiones.

5. Evalúe las expresiones que se muestran a continuación a través de sus correspondientes árboles de evaluación.

```
{
   double dres, d1 = 1, d2 = 2;

   int ires, i1 = 1, i2 = 2;

   dres = d1 + d2 / i2 * i1;

   dres = d1 + d2 / (i2 * i1);

   dres = d1 + d2 / i2 * i1 * 13 / 5;

   dres = d1 + d2 / (i2 * i1 * 13 / 5);

   dres = d1 + d2 / (i2 * i1 * 13 / 5.0);
}
```

6. En el listado que se muestra seguidamente determine qué elementos del estilo de código propuesto en I.2.7 han sido ignorados.

```
class C
{
   int ArrancóEn;
   int P;

void arranca()
{
ArrancóEn = Environment.TickCount;
}

void Para()
{
   P=Environment.TickCount;
}

   long tiempo()
   {
         return P-ArrancóEn;
   }
}
```

7. Refine la clase `Circunferencia` del presente capítulo con la incorporación de un constructor.

8. Diseñe e implemente una clase que dado un número de milisegundos entre 0 y 10^6, obtiene el equivalente en horas, minutos, segundos y centésimas de segundos.

9. Refine la clase `Cronómetro` de forma tal que el tiempo transcurrido se pueda mostrar en el formato horas, minutos, segundos y centésimas de segundos (auxíliese de la clase definida en el ejercicio anterior).

10. Refine la clase `Cronómetro` de forma tal que ahora se le pueda agregar una funcionalidad `Continuar`, que permita parar y luego seguir acumulando el tiempo (idea original en [24]).

11. La nota final de una asignatura se calcula a partir de las notas de los tres exámenes parciales, calculando la parte entera de la media de dichas notas. Diseñe e implemente una clase que permita representar asignaturas con su respectivo nombre, las tres notas de los parciales y la nota final.

12. Diseñe e implemente una clase que permita representar puntos de R^2, al menos garantice responsabilidades para mover dichos puntos y para determinar la distancia entre dos puntos.

13. Diseñe e implemente una clase que permita representar vectores de R^3, al menos garantice responsabilidades para las operaciones suma, producto por un escalar, producto escalar y producto vectorial.

14. Diseñe e implemente una clase que permita representar matrices cuadradas de orden 2, al menos garantice responsabilidades para las operaciones suma, producto y producto por un escalar.

15. Diseñe e implemente una clase que permita representar cuadriláteros, al menos garantice responsabilidades para calcular área, perímetro, longitud de las diagonales. Incorpore un método constructor y al menos una sobrecarga de este.

16. Diseñe e implemente una clase que permita representar triángulo, garantice al menos responsabilidades para calcular área, perímetro. Incorpore un método constructor y al menos una sobrecarga de este.

17. Diseñe e implemente una clase que permita representar rectas de la forma `y = mx + n`, garantice al menos responsabilidades para evaluar un valor, hallar su "cero", calcular la pendiente de una recta perpendicular a esta. Incorpore un método constructor y al menos una sobrecarga de este.

18. Diseñe e implemente una clase que permita representar medidas (expresadas en metros), garantice al menos responsabilidades necesarias para conocer el equivalente en `km, cm, mm`. Defina un método constructor para la misma.

19. Diseñe e implemente una clase que permita representar ecuaciones de segundo grado, garantice responsabilidades para calcular el determinante, calcular una de sus raices y obtener el resultado de evaluar un valor. Defina un método constructor para la misma.

20. Dadas las definiciones de las clases siguientes:

```
class Persona
{
   string nombre;
   int añoNacimiento;
   Persona (string nombre,int añoNacimiento)
   {
         this.nombre = nombre;
         this.añoNacimiento = añoNacimiento;
   }
   int Edad ( int unAño)
   {
         return unAño - añoNacimiento;
   }
}
```

```
class A
{
   int x,y;
   A (int unX, int unY)
   {
         x = unX;
         y = unY;
   }
   int Producto ()
   {
         return x*y;
   }
   int Suma ()
   {
         return x+y;
   }
}
```

```
class B
{
   string s1,s2;
   string Suma ()
   {
        return s1 + " " + s2;
   }
}
```

a). De las instrucciones que se listan a continuación ¿ Cuales son correctas son correctas y cuales no?. Justifique en cada caso.

```
{
    Persona juan = new Persona("juan", 1970), p , j;
    p.nombre = "paco";
    A valor1 = new A();
    j = juan;
    string s1 = valor1.Producto();
    B valor2 = new B("hola", "bien");
    B v1 = new B();
    v1.s1 = v1.s2 = "hola";
    j.nombre = "pepe";
    Persona p1 = j;
    j = new Persona("juan", 1780);
    j.añoNacimiento = 1910;
    p1.añoNacimiento = j.añoNacimiento ;
    v1.suma = "cadena";
}
```

b). Tomando en cuenta el listado anterior ¿Qué resultado devuelve las instrucciones siguientes?

```
juan.Edad();
juan.nombre;
```

I.2.13 Bibliografía complementaria

- Capítulos 2, 3 y 9 de Jesús García Molina et al., Introducción a la Programación, Diego Marín ,1999.
- Capítulos 2, 4, 5,7 y 8 de Jose A. González Seco, El lenguaje de programación C#, 2002. Puede descargarse en http://www.josanguapo.com/

Capítulo I.3

I.3 Aplicaciones simples en modo consola. Secuenciación

Pregunta: ¿Qué es posible realizar con los recursos presentados hasta el momento?

En los dos capítulos anteriores se han presentado algunos conceptos básicos del modelo OO, haciendo énfasis en el diseño e implementación de clases simples en un lenguaje de programación OO como C#. Es decir, hasta el momento es posible diseñar e implementar clases muy simples con métodos que utilicen la secuenciación como técnica básica para la implementación de algoritmos.

Ejemplo: Tomando como referencia la clase `Peso` descrita en el capitulo anterior, diseñar una aplicación que dado su peso en libras, permita obtener e imprimir el equivalente en kilogramos.

El algoritmo de solución a esta problemática al menos debe tener en cuenta la creación de una instancia de la clase `Peso` con su peso corporal pero muy poco es posible aportar además de esto, es decir: ¿Dónde se coloca esa secuencia de instrucciones?, ¿Cómo mostrar un mensaje en el monitor de la computadora?, etc.

Precisamente, tratando de encontrar respuesta a las interrogantes anteriores es que se desarrolla el presente capítulo. En el mismo se pretende presentar las secuencias de pasos para obtener aplicaciones muy simples en modo consola, utilizando algunas de las clases que se abordaron en el capítulo anterior.

En esencia, una aplicación en C# puede construirse a través de un conjunto de uno o más ficheros de código fuente con las instrucciones necesarias para que la aplicación funcione como se desea y que son pasados al compilador para que genere un programa en código ejecutable. Cada uno de estos ficheros no es más que un fichero de texto plano escrito usando caracteres Unicode y siguiendo la sintaxis propia de C#. Este código ejecutable resultante de la compilación se auxilia a su vez de otros ficheros que se denominan ensamblados y que serán analizados en detalle más adelante capítulo (V.I).

Uno de los elementos más importantes en el desarrollo de cualquier aplicación es la interfaz para la comunicación con el cliente o usuario. Existen diferentes tipos de interfases que se podrían agrupar en tres grupos generalizadores: interfaz en modo consola (ventana MSDOS), interfaz para ambiente Windows y por último las interfases para la Web. No es objetivo del presente texto adentrarse en las especificidades de las interfases avanzadas para la entrada y salida de los programas, es por ello que la atención es centrada en la creación de aplicaciones en modo consola donde la interacción se establece mediante la entrada y salida estándar de caracteres para lo cual C# dispone de primitivas muy simples y efectivas. Las aplicaciones de consola no tienen una interfaz de usuario y se ejecutan desde la línea de comandos.

Para la obtención de las aplicaciones se utiliza el ambiente integrado de desarrollo (IDE *Integrated Development Environment*) `Visual Studio .NET` (`VS.NET`), que es el ambiente de lujo para la construcción de aplicaciones sobre la plataforma `Microsoft .NET`. La decisión de apoyarse en este ambiente de trabajo se sustenta en el sinnúmero de facilidades que brinda dicho entorno y para no desviar la atención de los lectores hacia elementos superfluos que pueden entorpecer el desarrollo de los conceptos.

Para compilar una aplicación en `VS.NET` primero hay que incluirla dentro de algún proyecto. Los proyectos de `VS.NET` son contenedores que almacenan el material de desarrollo de una aplicación. Los proyectos de `VS.NET` contienen ficheros, carpetas y referencias a bases de datos, todos ellos necesarios durante la elaboración de un proyecto. Para desarrollar una aplicación en `VS.NET`, es necesario crear un `Nuevo Proyecto`, como se verá más adelante.

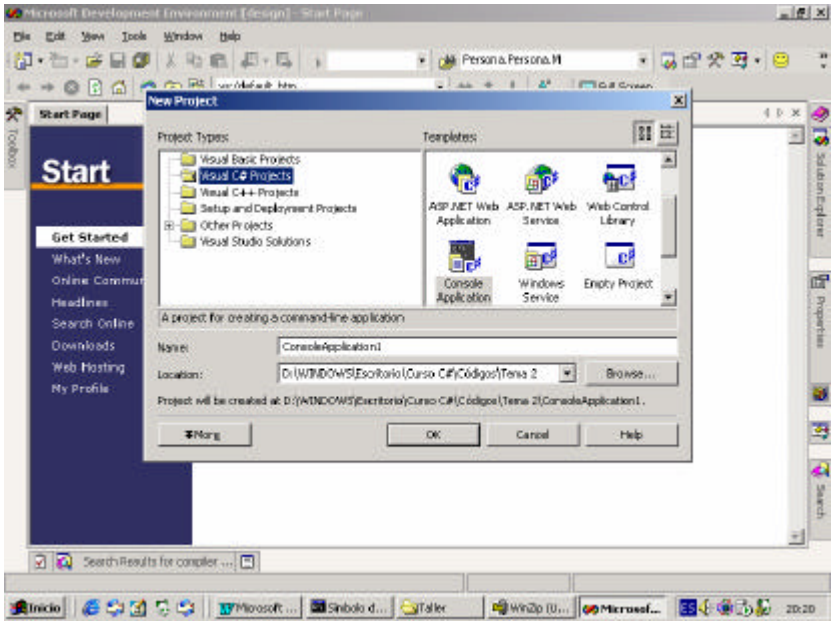
Los proyectos en `VS.NET` están contenidos dentro soluciones que facilitan la creación de aplicaciones simples o complejas mediante el empleo de plantillas y herramientas del entorno `VS.NET`. Una solución también puede contener múltiples proyectos u otras soluciones. En `VS.NET`, después de crear un proyecto este se coloca automáticamente dentro de una solución. El uso de soluciones y proyectos permite gestionar y organizar los archivos y carpetas que se emplean para crear las aplicaciones. Para conseguirlo `VS.NET` proporciona la ventana `Explorador de Soluciones` en la que se pueden ver y desde la que se pueden gestionar soluciones y proyectos.

I.3.1 Primera aplicación con Visual Studio .NET. Caso de estudio: `Hola Mundo`

Ejemplo: Utilizando `VS.NET` obtenga una aplicación consola que permita obtener el clásico mensaje `Hola Mundo` en el monitor de su computadora.

Para obtener esta primera aplicación se deben seguir los siguientes pasos:

Comenzar pulsando el botón **`New Project`** en la página de inicio que se muestra nada más que se ejecuta `VS.NET`, tras lo que se obtendrá una pantalla con el aspecto mostrado en la siguiente figura.



En el recuadro de la ventana mostrada etiquetado como **`Project Types`** se ha de seleccionar el tipo de proyecto a crear. Por supuesto, si se va a trabajar en C# la opción que habrá que escoger en la misma será siempre **`Visual C# Projects`**.

En el recuadro **`Templates`** se ha de seleccionar la plantilla correspondiente al subtipo de proyecto dentro del tipo indicado en **`Project Types`** que se va a realizar. Para obtener una aplicación consola, como es nuestro interés, hay que seleccionar el icono etiquetado como **`Console Application`**.

Por último, en el recuadro de texto **`Name`** se ha de escribir el nombre a dar al proyecto (**`HolaMundo`**) y en **`Location`** el del directorio base asociado al mismo. Nótese que bajo de **`Location`** aparecerá un mensaje informando sobre cual será el directorio donde finalmente se almacenarán los archivos del proyecto, que será el resultante de concatenar la ruta especificada para el directorio base y el nombre del proyecto.

Una vez tecleado el nombre del proyecto y determinada la localizacion presionamos el botón **`Aceptar`**. Al crear una aplicación de consola `VS.NET` le añade los archivos necesarios al `Proyecto`. Además, se crea un archivo de clase con el nombre **`Class1.cs`**. Este archivo contiene el código fuente que se muestra seguidamente:

```
using System;

namespace HolaMundo
{
    /// <summary>
    /// Descripción breve de Class1.
    /// </summary>
    class Class1
    {
```

```
            /// <summary>
            /// Punto de entrada principal de la aplicación.
            /// </summary>
            [STAThread]
            static void Main(string[] args)
            {
                  //
                  // TODO: agregar aquí código para iniciar la aplicación
                  //
            }
      }
}
```

A partir de la plantilla, que se muestra en el listado anterior, para obtener la clásica primera aplicación simplemente se tiene que escribir el código

```
Console.WriteLine("Hola Mundo")
```

dentro de la definición del método `Main(string[] args)` que también aparece en la referida plantilla.

Finalmente, para compilar y ejecutar tras ello la aplicación sólo hay que pulsar **`CTRL+F5`** o seleccionar **`Debug → Start Without Debugging`** en el menú principal de `Visual Studio.NET`. Para sólo compilar el proyecto, entonces hay que seleccionar **`Build → Rebuild All`**. De todas formas, en ambos casos el ejecutable generado se almacenará en el subdirectorio **`Bin\Debug`** del directorio del proyecto.

Pregunta: ¿Qué elementos le resultan novedosos en esta primera aplicación?

Vale la pena comenzar destacando el concepto de *punto de entrada* de una aplicación, este concepto se refiere precisamente a un método de nombre `Main` que contendrá el código por donde se ha de iniciar la ejecución de la misma.

Otro elemento novedoso es el contenedor **`namespace`** `HolaMundo`, nótese que esta nueva estructura sintáctica contiene a su vez una clase (**`class`** `Class1`) que es a quien pertenece el método `Main` donde colocamos la instrucción con anterioridad. Los *espacios de nombres* (*namespaces*) son precisamente los que permiten organizar los tipos de datos (clases), lo que por supuesto facilita su localización.

Los espacios de nombres también permiten poder usar en una misma aplicación varias clases con igual nombre si pertenecen a espacios diferentes. La idea es que cada fabricante defina sus tipos dentro de un espacio de nombres propio para que así no haya conflictos si varios fabricantes definen clases con el mismo nombre y se quieren usar a la vez en un mismo programa. Por supuesto, para que esto funcione no han de coincidir los nombres los espacios de cada fabricante, y una forma de conseguirlo es dándoles el nombre de la empresa fabricante, su nombre de dominio en Internet, etc.

Precisamente, esta es la forma en que se encuentra organizada la BCL, de modo que todas las clases más comúnmente usadas en cualquier aplicación pertenecen a un espacio de nombres llamado **`System`**, es por ello que aparece la instrucción **`using`** `System;` Debido a esto podemos colocar directamente el llamado a `Console.WriteLine("Hola Mundo")`, de no aparecer la referencia al **`namespace`** `System` a través del **`using`** es necesario hacer explícita entonces la presencia de la clase `Console` en dicho **`namespace`**, es decir: `System.Console.WriteLine("Hola Mundo")`.

Otro elemento de interés lo constituye el método `WriteLine`, este es un método de clase (II.2) que pertenece a la clase `Console`. Dicho método se emplea para escribir en la línea actual de la ventana de la consola. De la misma forma, para leer desde la ventana de la consola, es posible auxiliarse del método `Console.ReadLine`.

Sin dudas que existen otros elementos novedosos pero que aportan muy poco en este momento y que por tanto no se tratan en la respuesta a esta pregunta.

Ejemplo: Utilizando `VS.NET` obtener una aplicación consola que permita leer su nombre (`Laura` por ejemplo) y obtener el mensaje `Hola Laura` en el monitor de su computadora.

Para la obtención de esta aplicación también se siguen los pasos mostrados con anterioridad pero ahora se tiene que leer una cadena de caracteres antes, para luego imprimir el respectivo saludo como se muestra en el listado siguiente.

```
using System;

namespace I_3
{
    class Hola
    {
          static void Main(string[] args)
          {
                Console.Write("Teclee su nombre: ");
                string nombre = Console.ReadLine();
                Console.WriteLine("Hola " + nombre);

                Console.ReadLine();
          }
    }
}
```

Nótese en el listado anterior que `VS.NET` asigna una serie de identificadores a los diferentes recursos sintácticos que se sugiere cambiar por nombres nemotécnicos. En el listado anterior apenas se muestran los elementos imprescindibles para el buen funcionamiento de la aplicación solicitada con los respectivos cambios en los identificadores.

El algoritmo seguido en este caso ha sido: mostrar un mensaje para que teclee su nombre, luego leer el nombre tecleado y asignarlo a la variable `nombre` para más tarde concatenar este `nombre` con la cadena "`Hola`"; finalmente se hace nuevamente un llamado a `Console.WriteLine` para que la aplicación quede a la espera de un `enter` antes de finalizar (pruebe a ejecutar desde fuera del entorno `Visual Studio` la aplicación sin colocar esta última instrucción y comprobará lo que sucede).

Queda a su consideración la variante de implementación del método `Main` que se muestra en el siguiente listado.

```
static void Main(string[] args)
{
      Console.Write("Teclee su nombre: ");
      Console.WriteLine("Hola " + Console.ReadLine());
      Console.ReadLine();
}
```

I.3.1.1 Modularidad

El concepto de *módulo* no es exclusivo del modelo OO, pero encuentra en el mismo un buen soporte y aplicación. Particularmente el concepto de `namespace` es un ejemplo muy representativo del concepto de módulo, es por ello que se le dedica de manera muy breve esta sección a este último concepto.

"En un sentido los módulos pueden considerarse simplemente como una técnica mejorada para crear y manejar espacios de nombres" [1] (*namespaces*). En este sentido una aplicación importante del concepto de módulo es para la creación de bibliotecas de clases, es decir, un espacio de nombres que va a contener diferentes clases. En el sentido más amplio, un módulo es un "contenedor" de código que permite mejorar la estructura y organización de las aplicaciones de software.

Una clara aplicación del concepto de módulo puede verse a través de las propias clases. "Una clase será a su vez un módulo y un tipo. Como módulo la clase encapsula (encierra) un número de facilidades o recursos que ofrecerá a otras clases (sus clientes). Como tipo describe un conjunto de objetos, instancias o ejemplares que existirán en tiempo de ejecución. La conexión entre estos dos enfoques es muy simple: los recursos

ofrecidos son precisamente las operaciones que llevarán a cabo las instancias del tipo. La POO consiste en definir tipos o clases de objetos que cumplan determinadas propiedades y que puedan llevar a cabo una determinada funcionalidad o comportamiento" [3].

En el caso particular de las diferentes versiones de Turbo Pascal y Object Pascal por ejemplo, el concepto de módulo se implementa también a través de las *units* mientras que en el caso particular de la plataforma Microsoft .NET cobra entonces vital importancia el concepto de *ensamblado* (*assembly*) (V.1) también como ejemplo de aplicación del concepto de módulo. Es válido destacar que en el caso de los ensamblados de .NET como ejemplo de módulo, pueden contener diferentes espacios de nombres e incluso entre éstos también puede haber anidamiento, es decir, un *namespace* puede contener a su vez otro *namespace*.

Sin dudas, son disímiles las aplicaciones posibles a mencionar del concepto de módulo como por ejemplo el método como contenedor de código que representa funcionalidades de las clases y objetos.

Resumiendo, la modularidad permite dividir la solución de un problema en partes más pequeñas y dar solución a cada una de éstas de forma independiente para después integrarlas como un único programa ejecutable.

I.3.2 Inclusión de referencias

Ejemplo: Tomando como referencia la clase `Peso` implementada en al capítulo anterior anterior, diseñe una aplicación que dado su peso en libras, permita obtener e imprimir el equivalente en kilogramos.

Para la obtención de esta aplicación se trabaja con un peso arbitrario. En términos de algoritmo de solución apenas se tiene que crear un objeto (instancia) de `Peso` con el correspondiente valor en libras y enviarle un mensaje para que retorne la correspondiente conversión a kilogramos que inmediatamente se imprime a través de la primitiva `Console.WriteLine()`.

Ejemplo: En la respuesta al ejercicio anterior se le ha pasado muy por encima a un detalle: ¿Dónde se coloca el código fuente de la clase `Peso`?

Realmente el código fuente de la clase `Peso` podría colocarse en el mismo fichero de la aplicación pero esta no es una buena práctica ya que no contribuye al posterior reuso de esta clase, ni a la organización más efectiva de los códigos.

Para la implementación de las clases se recomiendan las *bibliotecas de clases*, estas no son más que ficheros donde pueden coexistir varias clases (inclusive diversos espacios de nombres). Estos ficheros una vez que se compilan producen los denominados *ensamblados* (*assemblies*, archivos con extensión `dll`) que se analizan en V.1. Para la creación de las bibliotecas de clases se pueden seguir los mismos pasos que para la creación de las aplicaciones consola pero llegado el recuadro **`Templates`** se ha de seleccionar la plantilla correspondiente al subtipo de proyecto dentro del tipo indicado en **`Project Types`** que se va a realizar; en este caso hay que seleccionar el icono etiquetado como **`Class Library`**. Al concluir con este asistente se genera un fichero con la estructura que se muestra en el listado siguiente:

```
using System;
namespace Peso
{
    /// <summary>
    /// Descripción breve de Class1.
    /// </summary>
    public class Class1
    {
          public Class1()
          {
                //
                // TODO: agregar aquí la lógica del constructor
                //
          }
    }
```

```
}
```

Nótese que se tiene una plantilla para la creación de una clase, en esta plantilla por defecto se tiene un `namespace` que siempre se le asigna el nombre del `Proyecto` y que contiene a la clase de la plantilla que por defecto se le asigna el identificador `Class1`. En el listado siguiente se muestra una variante simplificada de la clase `Peso` incorporada a la plantilla generada con anterioridad, en la cual se ha cambiado el nombre del **`namespace`** por `I_2` y el de `Class1` por `Peso`.

```
using System;
namespace I_2
{
    class Peso
    {
          double libras;
          public Peso(double unPesoLibras) {...}
          public double Kilogramos(){...}
          public double Gramos() {...}

    }
}
```

Nota: Como estilo de código del texto se irán denotando los espacios de nombres con identificadores semejantes a los respectivos capítulos donde sean definidas las clases.

Ejemplo: Otro detalle ignorado hasta el momento: ¿Cómo es posible tener acceso a este código para definir sus respectivas instancias?

Para responder a esta pregunta debemos tener en cuenta la inclusión de referencias a los respectivos ensamblados. Es decir, el fichero que contiene el código fuente (aplicación o ensamblado) necesita utilizar recursos que están en otro fichero (ensamblado); esto se traduce en que el primero tiene que establecer una referencia con el segundo.

Para añadir a un proyecto referencias a ensamblados externos basta seleccionar **`Project → Add Reference`** en el menú principal de `VS.NET` (existen otras formas que con la practica iremos dominando). Ahora si se tienen todos los elementos para obtener la aplicación consola que se muestra en el listado siguiente:

```
using System;
using I_2;
namespace I_3
{
      class MiPeso
      {
            static void Main(string[] args)
            {
                  Peso miPeso = new Peso(165);
                  Console.WriteLine(" Mi peso en kg es "  +
                                    miPeso.Kilogramos());
                  Console.ReadLine();
            }
      }
}
```

Pregunta: ¿Es imprescindible la inclusión de la instrucción **`using`** `I_2;` aún cuando previamente fue establecida la referencia?

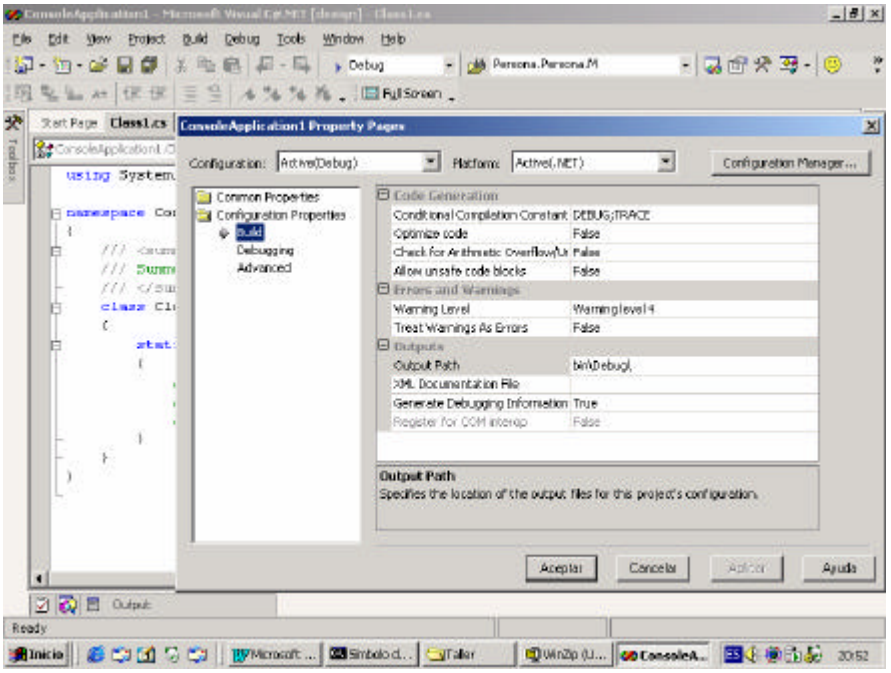
Por la forma en que se ha hecho referencia a la clase `Peso` sí, pues la aplicación consola se ha definido en un espacio de nombres diferente al de la clase `Peso`. En estas circunstancias si no se incluye la cláusula **`using`** `I_2;` entonces se tendrían obligatoriamente que hacer referencia a la clase `Peso` de la siguiente

forma: I_2.Peso. En el caso de que ambas clases coincidan en el espacio de nombres no es necesario incluir el **`using`** ni el *namespace* como prefijo de la clase.

Nota: A partir de ahora se mostrará apenas el código del método `Main` cuando se desarrollen las aplicaciones.

I.3.2.1 Explorador de soluciones

En el extremo derecho de la ventana principal de `Visual Studio.NET` es posible encontrar el denominado *Explorador de Soluciones* (**`Solution Explorer`**) (si no lo encuentra, puede seleccionar **`View → Solution Explorer`**), que es una herramienta que permite consultar cuáles son los archivos que forman el proyecto. Si selecciona en él el icono correspondiente al proyecto en que estamos trabajando y pulsa **`View → Property Pages`** obtendrá una hoja de propiedades del proyecto con el aspecto mostrado en la figura siguiente.



A través de esta ventana es posible configurar de manera visual la mayoría de opciones del Proyecto. Por ejemplo, para cambiar el nombre del fichero de salida se indica su nuevo nombre en el cuadro de texto **`Common Properties → General → Assembly Name`**; para cambiar el tipo de proyecto a generar se utiliza **`Common Properties → General → Output Type`**; y el tipo que contiene el punto de entrada a utilizar (método `Main`) se indica en **`Common Properties → General → Startup Object`**.

Otra de las utilidades importantes del Explorador de Soluciones es para establecer manipular las referencias. Verifique que cuando se visualiza la ventana **`Solution Explorer`** aparece el icono **`References`** dentro de cada proyecto; a través de este icono es posible agregar o eliminar referencias de forma muy rápida.

I.3.3 Conversión de valores

Ejemplo: Obtener una aplicación consola donde se lea un peso cualquiera en libras y se obtenga su equivalente en kilogramos.

Para la solución de este ejercicio, en principio apenas es necesario el auxilio de la primitiva `Console.ReadLine` para leer el valor que se desea convertir. Sin embargo, aparece una nueva problemática: el método `ReadLine` retorna una cadena de caracteres (**`string`**) y el constructor de la clase `Peso` necesita un valor real (**`double`**). Es decir, es necesario hacer una conversión de **`string`**

para **double**; de ello se ocupará el método `Parse` de la clase **double** (de forma general todas las clases que representan dominios de valores numéricos tienen su propio método `Parse`). En el listado siguiente (en un mismo bloque de instrucciones) se muestran dos variantes equivalentes de solución de este ejercicio.

```
{
    Console.Write("Teclee un peso en libras: " );
    double unPesoLibras = double.Parse(Console.ReadLine());
    Peso unPeso = new Peso(unPesoLibras);
    Console.WriteLine(" Conversión a kg " +
                      unPeso.Kilogramos().ToString());
    Console.WriteLine();

    Console.Write("Teclee otro peso en libras: " );
    unPeso = new Peso(double.Parse(Console.ReadLine()));
    Console.WriteLine(" Conversión a kg " +
                      unPeso.Kilogramos().ToString());
    Console.ReadLine();
}
```

Ejemplo: Obtener una aplicación donde sean leídos 2 valores enteros y se calcule el promedio de los mismos.

Un posible algoritmo de solución en este caso consiste en leer los 2 valores enteros, luego sumarlos y finalmente dividir la respectiva suma entre 2. En el listado siguiente se muestra una variante de implementación de este algoritmo.

```
{
    Console.Write("Teclee el primer valor: ");
    int v1 = int.Parse(Console.ReadLine());

    Console.Write("Teclee el primer valor: ");
    int v2 = int.Parse(Console.ReadLine());

    double promedio = (v1 + v2) / 2;
    Console.WriteLine(" Promedio = " + promedio.ToString());
}
```

Ejemplo: Ejecutar el código presentado en el listado anterior para los valores 1 (`v1`) y 2 (`v2`). ¿Cuál será el resultado que se imprime?

El resultado que se imprime es 1, esto esta dado porque en la expresión para el cálculo del promedio solamente intervienen valores enteros; por lo tanto la división que se realiza es la división entera.

Del capítulo anterior se conoce que expresando alguno de los términos como un literal real entonces se obtiene realmente la división con decimales como se muestra a continuación:

```
promedio = (v1 + v2) / 2.0;
```

Ejemplo: Imagine ahora que no se tienen valores literales en la expresión. ¿Cómo podríamos obtener una expresión similar la anterior?

En este caso se necesita hacer una conversión de tipos más conceptual denominada *casting* y que será abordada en detalle en la Parte II. Esta conversión de tipos se puede realizar entre tipos *equivalentes* como es el caso de **int** y **double**, esta equivalencia está dada porque todo valor **int** es a su vez un valor **double** y por ello todo valor **int** puede ser utilizado como un **double** (y no viceversa). La sintaxis de esta nueva forma de conversión se muestra a continuación y se aplica en varias oportunidades en el listado posterior:

```
(<tipo de dato>) <expresión>
```

```
{
    const int dos = 2;
```

```
        promedio = ((double)v1 + v2) / dos;
        promedio = (v1 + (double)v2) / dos;
        promedio = (double)(v1 + v2) / dos;
        promedio = (v1 + v2) / (double)dos;
    }
```

I.3.4 Recolección automática de basura

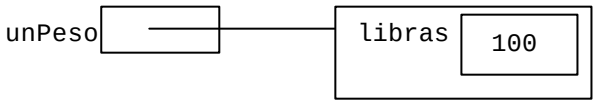
Situación de análisis

Seguidamente se muestra una variante simplificada de la aplicación que en dos formas diferentes lee un valor de peso en libras y lo convierte en kilogramos (a través de una misma variable `unPeso`):

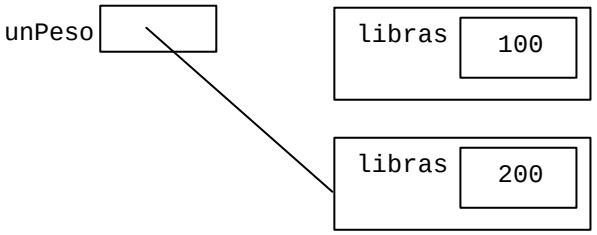
```
{
      double unPesoLibras = double.Parse(Console.ReadLine());
      Peso unPeso = new Peso(unPesoLibras);
      Console.WriteLine(" Conversión a kg " +
                        unPeso.Kilogramos().ToString());

      unPeso = new Peso(double.Parse(Console.ReadLine()));
      Console.WriteLine(" Conversión a kg " +
                        unPeso.Kilogramos().ToString());
}
```

Pregunta: ¿Qué sucede con el primer objeto al que hacía referencia la variable `unPeso`?

Primeramente se asume que en la primera oportunidad fue tecleado el valor 100 y en la segunda 200. Para comprender mejor la problemática, se realiza una simulación de lo que sucede en la memoria con la variable `unPeso` y los diferentes objetos a través de la figura siguiente.



a) Después de la creación del primer objeto `Peso`



b) Después de la creación del segundo objeto `Peso`

Situación de análisis

Nótese que ha quedado un objeto en la memoria que no es referenciado por ninguna variable: ¿Será posible que se esté dejando *basura* en la memoria?

Pregunta: ¿Cómo es posible eliminar de memoria los objetos que no están siendo referenciados o de forma general los que no interesa utilizar?

En C# no hay que preocuparse por esto como lo tendría que hacer en otros lenguajes. C# dispone de un mecanismo de *recolección automática de basura*.

C# se auxilia de un recolector de basura (*garbage collector*) que se incluye en el CLR (V.I) que evita que el programador deba tener en cuenta cuándo ha de destruir los objetos que dejen de serle útiles. Este recolector es un recurso que se activa cuando se quiere crear algún objeto nuevo y se detecta que no queda memoria libre para hacerlo, caso en que el recolector recorre la memoria dinámica asociada a la aplicación, detecta qué objetos hay en ella que no puedan ser accedidos por el código de la aplicación, y los elimina para limpiar la memoria de "objetos basura" y permitir la creación de otros nuevos. Gracias a este recolector se evitan errores de programación muy comunes como intentos de borrado de objetos ya borrados, agotamiento de memoria por olvido de eliminación de objetos inútiles o solicitud de acceso a miembros de objetos ya destruidos.

I.3.5 Caso de estudio: aplicación con instancias de la clase Persona

Ejemplo: Utilizando la clase `Persona` diseñada en I.2 obtener una aplicación que en una única variable permita crear dos instancias de dicha clase y muestre los respectivos mensajes con la correspondiente edad.

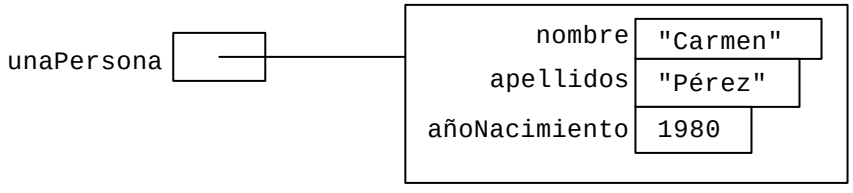
La secuencia de pasos para obtener esta aplicación sería leer cada uno de los datos que necesita el constructor de la clase `Persona` para crear sus instancias (nombre, apellidos y año de nacimiento), luego crear la primera instancia de dicha clase y se le envía un mensaje para obtener su edad e imprimirla; en el caso de la segunda instancia se repite la misma secuencia de pasos. Una variante de implementación de esta aplicación se muestra en el listado siguiente:

```
{
    Console.WriteLine("Datos de la primera persona");
    Console.Write("Nombre: ");
    string nombre = Console.ReadLine();
    Console.Write("Apellidos: ");
    string apellidos = Console.ReadLine();
    Console.Write("Año de nacimiento: ");
    uint añoNacimiento = uint.Parse(Console.ReadLine());
    Persona unaPersona = new Persona(nombre, apellidos, añoNacimiento);
    Console.WriteLine("Su edad es: " + unaPersona.Edad().ToString());
    Console.WriteLine();

    Console.WriteLine("Datos de la segunda persona");
    Console.Write("Nombre: ");
    nombre = Console.ReadLine();
    Console.Write("Apellidos: ");
    apellidos = Console.ReadLine();
    Console.Write("Año de nacimiento: ");
    añoNacimiento = uint.Parse(Console.ReadLine());
    unaPersona = new Persona(nombre, apellidos, añoNacimiento);
    Console.WriteLine("Su edad es: " + unaPersona.Edad().ToString());
    Console.ReadLine();
}
```

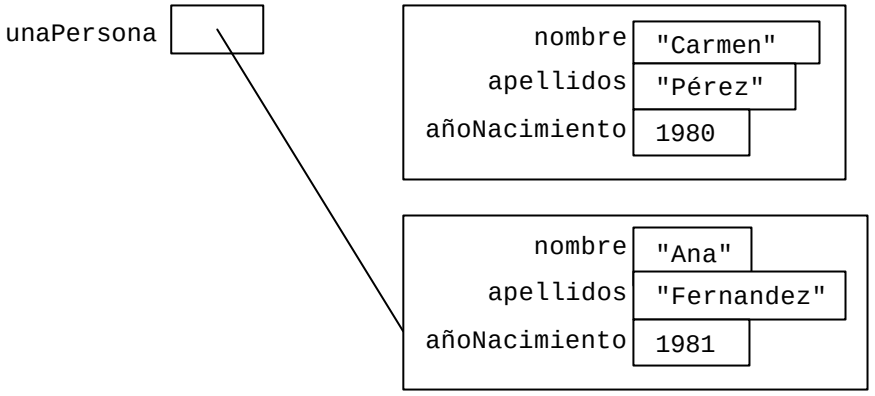
Sin dudas que se han presentado dos segmentos de código muy parecidos que con certeza es posible refinar y mejorar y de hecho se realizará en el capítulo II.3.

Ejemplo: Ilustre gráficamente (de forma semejante a como se hace en la sección anterior con la variable `unPeso`) qué sucede con la variable `unaPersona` del listado anterior.

Para comprender mejor la ilustración se asume que la primera instancia es creada con el juego de valores <`"Carmen", "Pérez", 1980`> y la segunda <`"Ana", "Fernández", 1980`>, entonces en la siguiente figura se hace una simulación de lo que sucede en la memoria con la variable `unaPersona`.



a) Después de la creación de la primera instancia



b) Después de la creación de la segunda instancia

I.3.6 Caso de estudio: aplicación con instancia de la clase Cronómetro

Ejemplo: Utilizando la clase `Cronómetro` (I.1.13), obtener una aplicación que determine que tiempo una persona se demora en teclear su nombre (idea original en [24]).

El algoritmo de solución en este caso consiste en enviarle un mensaje por pantalla al usuario para que teclee su nombre, luego se crea una instancia de `Cronómetro` y se le envía un mensaje para que eche a andar y se espera a que el usuario teclee su nombre; en cuanto termina de teclear su nombre, entonces se le envía otro mensaje al objeto `Cronómetro` para que pare y finalmente se le envía un último mensaje para que nos muestre el tiempo para imprimirlo. Una variante de este algoritmo se muestra en el listado siguiente:

```
{
    Console.Write("Teclee su nombre: ");
    Cronómetro c = new Cronómetro();
    c.Arranca();

    string nombre = Console.ReadLine();
    c.Para();

    Console.WriteLine("{0} usted demoró {1} mseg en teclear su nombre",
                      nombre, c.Tiempo().ToString());
    Console.ReadLine();
}
```

Es novedoso en este código la parametrización de la cadena a imprimir ({0} y {1}) en el último llamado al método `WriteLine`, nótese que inmediatamente después de la cadena aparecen dos parámetros que son precisamente los que sustituyen a {0} y {1} respectivamente.

I.3.7 Ejercicios

1. Obtener una aplicación que permita obtener el área de una circunferencia de centro en (0,0) y radio 10 (auxíliese de la clase `Circunferencia` como solución al ejercicio I.2.12.5).

2. Obtener una aplicación que permita calcular el área de dos circunferencias a partir de las coordenadas de sus respectivos centros y radios (auxíliese de la clase `Circunferencia` del capitulo anterior). Utilice una única variable de tipo `Circunferencia` y luego ilustre gráficamente lo que sucede con esta variable para un par de juegos de valores.

3. Obtenga una aplicación que permita obtener la temperatura de ebullición del agua (100 grados Celsius) en grados Fahrenheit (auxíliese de la clase `Temperatura` del capítulo anterior).

4. Obtenga una aplicación que dado un valor de temperatura en grados Celsius permita obtener su equivalente en grados Fahrenheit.

5. Obtener una aplicación que dada una cantidad de dinero permita obtener su desglose (mínimo) en los billetes existentes en Cuba (auxíliese de la clase `Dinero` del capítulo anterior).

6. Modifique el listado de la seccion I.3.6 de forma tal que ahora se pueda obtener la salida en el formato horas, minutos, segundos y centésimas de segundos (auxíliese de la solución al ejercicio I.2.12.7 del capitulo anterior).

7. Modifique el listado de la seccion I.3.6 de forma tal que se pueda tomar en dos oportunidades cuánto se demora el usuario en teclear su nombre y se imprima el tiempo promedio en milisegundos que ha consumido (auxíliese de la solución al ejercicio I.2.12.7) (idea original en [24]).

8. Obtenga una aplicación que permita crear una instancia de la clase definida como solución a I.2.12.9 que imprima la respectiva nota final. También refine la clase definida en I.2.12.9 de forma tal que la nota final sea el valor entero más cercano a la media de los tres exámenes parciales y no la parte entera de dicha media.

9. Diseñe una aplicación que permita crear dos puntos de R^2 e imprima la distancia entre ambos (auxíliese de la solución al ejercicio I.2.12.10).

10. Diseñe una aplicación que permita crear dos vectores de R^3 e imprima los resultados de las operaciones suma, producto escalar y producto vectorial entre ambos; así como el producto de un escalar por cada uno de ellos (auxíliese de la solución al ejercicio I.2.12.11).

11. Diseñe una aplicación que permita crear dos matrices cuadradas de orden 2 e imprima los resultados de las operaciones suma y producto entre ambas; así como el producto de un escalar por cada una de ellas (auxíliese de la solución al ejercicio I.2.12.12).

I.3.8 Bibliografía complementaria

- Capítulos 2 de Jesús García Molina et al., Introducción a la Programación, Diego Marín ,1999.
- Capítulos 2, 3 y 10 de Jose A. González Seco, El lenguaje de programación C#, 2002. Puede descargarse en http://www.josanguapo.com/

Tema II

II. ANÁLISIS DE CASOS. ENCAPSULAMIENTO

Objetivos

- **Diseñar** algoritmos que necesiten del análisis de casos como técnica de descomposición.
- **Utilizar** estructuras de control alternativas.
- **Definir** clases con componentes que a su vez son objetos.
- **Garantizar** la integridad de los objetos a través del concepto de encapsulamiento.
- **Caracterizar** el concepto de Programación por Contratos.
- **Lanzar** excepciones.
- **Capturar** excepciones.
- **Obtener** aplicaciones en modo consola que involucren instancias de las clases definidas en el desarrollo del tema y los conceptos relacionados.

Contenido

II.1 Análisis de casos. Estructuras de control alternativas.
II.2 Componentes que a su vez son objetos.
II.3 Encapsulamiento. Programación por contratos. Excepciones.

Descripción

El tema comienza con la presentación de situaciones de análisis que necesitan de estructuras de control alternativas para su solución. (análisis de casos).

En un segundo momento se muestran situaciones de análisis que requieren de la relación de composición entre las clases, es decir, definiremos componentes de las clases que a su vez son objetos.

Finalmente se profundiza en el concepto de Encapsulamiento y con este se muestran los elementos esenciales de la Programación por Contratos y el tratamiento más elemental de las excepciones.

El objetivo integrador de este tema es obtener aplicaciones en modo consola que involucren los contenidos presentados en el tema .

Capítulo II.1

II.1 Estructuras de control alternativas. Análisis de casos

Los algoritmos que se han desarrollado hasta el momento se desarrollan a través de un flujo secuencial, es decir, las instrucciones se ejecutan una a continuación de la otra en el mismo orden que son escritas.

Sin embargo, con mucha frecuencia, en la solución de los problemas aparecen situaciones en que la ejecución de una instrucción depende de si se cumple o no una situación. Es por ello que en los lenguajes de programación aparecen las *estructuras de control alternativas* o *condicionales* que son instrucciones que permiten ejecutar bloques de código sólo si se pone de manifiesto una determinada condición. Estas estructuras son las que permiten resolver problemas a través de la técnica de descomposición denominada *análisis de casos* que es precisamente el objetivo de este capítulo.

II.1.1 Estructura de control alternativa simple (`if`)

Para el desarrollo de esta sección se continúa desarrollando el caso de estudio `Universidad` presentado en el capítulo I.1.

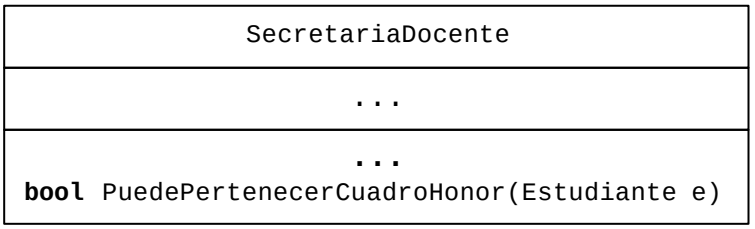
Situación de análisis

Próximo al mediodía el decano de la facultad de informática llama por teléfono a la secretaria docente y le pregunta si el estudiante `juan` puede pertenecer al Cuadro de Honor de la facultad. Poco después la secretaria docente se encarga de localizar a `juan` para preguntarle su índice académico e inmediatamente este le responde (4.83). Minutos después Ana es entonces quien llama por teléfono al decano para comunicarle que `juan` cumple con la condición primaria para pertenecer al Cuadro de Honor.

Pregunta: ¿Cuál fue el criterio seguido por la secretaria docente para responderle afirmativamente al decano?

Simplemente determinar que el promedio de `juan` es superior a 4.75, que es la condición primaria para pertenecer al Cuadro de Honor. A esta responsabilidad del objeto `informática.secretariaDocente` (y de todas las instancias de la referida clase `SecretariaDocente`) se le puede denominar `PuedePertenecerCuadroHonor`.

A continuación se presenta el diagrama con la inclusión de la nueva responsabilidad en la clase `SecretariaDocente` y seguidamente se muestra como el objeto `informática.secretariaDocente` utiliza la responsabilidad `PuedePertenecerCuadroHonor` para determinar si `juan` cumple con la condición primaria para pertenecer al Cuadro de Honor




```
informática.secretariaDocente.PuedePertenecerCuadroHonor(juan)
```

Pregunta: ¿Cómo implementar este nuevo método de la clase `SecretariaDocente`?

Un algoritmo para implementar este método podría ser enviarle un mensaje a `juan` para que informe su promedio y luego el resultado compararlo con 4.75; si es mayor o igual se retorna valor verdadero (**`true`**) y en otro caso falso (**`false`**). La implementación del referido método se muestra a continuación. Este método establece una relación de uso entre las clases `SecretariaDocente` y `Estudiante`.

```
public class SecretariaDocente
{
    public bool PuedePertenecerCuadroHonor(Estudiante e)
    {
          bool temp = false;
          if (e.Promedio() >= 4.75)
                temp = true;
          return temp;
    }
}
```

Note que la implementación que se muestra con anterioridad no coincide exactamente con el algoritmo descrito. El algoritmo que se ha implementado en este caso ha sido el siguiente: se inicializa una variable temporal `temp` con el valor `falso` (**`false`**), posteriormente se evalúa la condición de la instrucción **`if`**, (`e.Promedio() >= 4.75`) y solamente si el resultado es **`true`** es que entonces cambia el valor de la variable `temp` por **`true`**, finalmente se retorna el valor contenido en la variable `temp`.

Analizando el código anterior es posible determinar la presencia de una nueva instrucción o estructura sintáctica del lenguaje, en este caso la *estructura de control alternativa simple* **`if`**. La sintaxis de esta nueva estructura sintáctica presente en el código es la siguiente:

```
if (<condición>)
      <bloque de instrucciones>;
```

La semántica de esta instrucción es la siguiente: se evalúa la expresión lógica `<condición>`, si el resultado de su evaluación es verdadero (`true`) se ejecuta el `<bloque de instrucciones >`. La instrucción condicional **`if`** refleja que en un punto dado del programa se quiere ejecutar alternativamente (en dependencia de la evaluación de una condición), un conjunto de instrucciones.

A continuación se muestra otra implementación de método `PuedePertenecerCuadroHonor` equivalente a la anterior.

```
public class SecretariaDocente
{
      public bool PuedePertenecerCuadroHonor(Estudiante e)
      {
            if (e.Promedio() >= 4.75)
                  return true;
            return false;
      }
}
```

En este caso, el algoritmo implementado ha sido el siguiente: evaluar la condición de la instrucción **`if`**, (`e.Promedio() >= 4.75`) y si el resultado es `verdadero`, entonces retorna **`true`** , solamente si el resultado de la evaluación es `falso` es que se retorna el valor **`false`**. El funcionamiento de este bloque de código se garantiza porque la ejecución de la instrucción **`return`** permite terminar inmediatamente con la ejecución del método en cuestión.

Esta última implementación refleja fielmente el primer algoritmo propuesto e incluso aún existe una variante que lo refleja con mayor fidelidad como se muestra a continuación.

```
public class SecretariaDocente
{
      public bool PuedePertenecerCuadroHonor(Estudiante e)
      {
            if (e.Promedio() >= 4.75)
                  return true;
            else
                  return false;
```

```
    }
}
```

Analizando el código anterior es posible determinar la presencia de una variante de la estructura alternativa **if** ahora con la presencia de una nueva cláusula (**else**). La sintaxis de esta variante sintáctica de la estructura **if** se puede expresar de la siguiente forma:

```
if (<condición>)
      <bloque de instrucciones 1>;
else
      <bloque de instrucciones 2>;
```

La semántica de esta variante es la siguiente: El `<bloque de instrucciones 1>` se ejecuta en caso de que se cumpla la `<condición>`, en otro caso se ejecuta entonces el `<bloque de instrucciones 2>`.

Al comparar las diferentes variantes presentadas en esta sección es posible concluir que la estructura de control alternativa **if** puede ser descrita de la siguiente forma (dado el carácter opcional de la cláusula **else**):

```
if (<condición>)
      <bloque de instrucciones 1>;
[else
      <bloque de instrucciones 2>;]
```

Nota: Vale aclarar que un bloque de instrucciones puede comenzar a su vez con otro **if**.

Situación de análisis

Analizar el listado siguiente:

```
public class SecretariaDocente
{
      public bool PuedePertenecerCuadroHonor(Estudiante e)
      {
            if (e.Promedio() >= 4.75)
                  return true;
      }
}
```

Pregunta: ¿Cuál será el resultado de la compilación del listado anterior desde el punto de vista sintáctico y semántico?

Desde el punto de vista sintáctico todo está bien porque todas las estructuras sintácticas están correctamente escritas. Sin embargo, desde el punto de vista semántico existe un problema porque cuando el resultado de la evaluación de la expresión es **false**, entonces no se retorna valor alguno. Algunos compiladores son poco exigentes con esto y apenas retornan un mensaje de advertencia, por lo que sin dudas en la ejecución de la aplicación es donde aparecerían los errores, que generalmente son más difíciles de determinar. Sin embargo, el compilador de C# es intransigente con esto y nos informa de un error semejante al siguiente: `no todas las rutinas de código devuelven un valor`.

Una implementación interesante y que explota al máximo de las posibilidades la combinación del **return** con las expresiones lógicas se muestra a continuación.

```
public class SecretariaDocente
{
      public bool PuedePertenecerCuadroHonor(Estudiante e)
      {
            return e.Promedio() >= 4.75;
      }
}
```

En este caso el método retorna directamente un valor en función de la expresión, es decir si es satisfactoria la evaluación de la expresión el método retorna valor **`true`** y en caso contrario retorna **`false`**. Sobre este tipo de construcción se insistirá en lo adelante por considerarla una buena práctica de programación.

Aún cuando la mejor solución no es a través de un **`if`**, se ha utilizado el ejemplo anterior para de forma muy simple realizar este primer análisis de casos y mostrar la estructura de control alternativa simple. De todas formas no hay por qué preocuparse ya que en las siguientes secciones se profundiza en algoritmos que necesiten del análisis de casos como técnica de descomposición para el diseño de algoritmos.

II.1.2 Análisis de casos

Situación de análisis

El análisis de casos consiste en dividir el dominio de datos en un conjunto de subdominios, cada uno de los cuales determina "un caso" que lleva asociada la ejecución de una o varias acciones. Los subdominios se expresan mediante expresiones lógicas y deben ser mutuamente excluyentes entre sí. A partir del estado inicial se alcanzará el final ejecutando la acción correspondiente a la única expresión lógica verdadera [9].

Ejemplo: Obtener una aplicación consola que permita leer dos números enteros (`x` e `y`) e imprima el mayor (`mayor`).

El algoritmo de solución puede dividirse en tres etapas: leer los números, obtener el mayor y posteriormente imprimirlo, como se muestra a continuación:

```
{
    int x = int.Parse(Console.ReadLine());
    int y = int.Parse(Console.ReadLine());
    int mayor;
    if (x >= y)
          mayor = x;
    else
          mayor = y;
    Console.WriteLine("Mayor = {0}", mayor);
}
```

En una segunda variante es posible unificar las dos últimas acciones como se muestra seguidamente:

```
{
    int x = int.Parse(Console.ReadLine());
    int y = int.Parse(Console.ReadLine());
    if (x >= y)
          Console.WriteLine("Mayor = {0}", x);
    else
          Console.WriteLine("Mayor = {0}", y);
}
```

Note como para calcular el mayor elemento de x e y se puede realizar un análisis de casos dividiendo el dominio de datos en dos subdominios: `x >= y` e `x < y`. Para aquellos datos que pertenezcan al primer dominio se realizara la acción `mayor = x` y para el resto `mayor = y`.

Nota: La cláusula **`else`** en su semántica de exclusión con relación a su respectivo **`if`** da la posibilidad de ignorar el chequeo del último de los subdominios.

Situación de análisis

Analice la variante para imprimir el mayor entre dos números enteros que se propone en el listado siguiente y realice corridas para diferentes juegos de datos.

```
{
      int x = int.Parse(Console.ReadLine());
      int y = int.Parse(Console.ReadLine());
      if (x > y)
            Console.WriteLine("Mayor = {0}", x);
      else
            if (x < y)
                  Console.WriteLine("Mayor = {0}", y);
}
```

Pregunta: ¿Qué sucede cuando los valores de `x` e `y` son iguales?

No se imprime ningún valor.

Esto ha sucedido porque se han olvidado casos, esto quiere decir que existen entradas de datos que no se consideran en ninguno de los subdominios pues como se puede apreciar no se tiene en cuenta el estado inicial en el que `x` e `y` son iguales. El *olvido de casos* es uno de los errores típicos que puede cometerse en el momento de dividir el dominio de datos en subdominios.

Situación de análisis

Analice ahora la variante para imprimir el mayor entre dos números enteros que se propone en el listado siguiente y también realice corridas para diferentes juegos de datos.

```
{
      int x = int.Parse(Console.ReadLine());
      int y = int.Parse(Console.ReadLine());

      if (x >= y)
            Console.WriteLine("Mayor = {0}", x);
      if (x <= y)
            Console.WriteLine("Mayor = {0}", y);
}
```

Pregunta: ¿Qué sucede cuando los valores de `x` e `y` son iguales?

Ahora se imprime dos veces el valor mayor.

La definición de casos que no son mutuamente excluyentes es otro error que se puede cometer en la determinación de los subdominios. En este caso, existiría un solapamiento de algunos subdominios, dándose el caso de que para un dato de entrada se puedan cumplir varios casos. En el listado anterior se muestra un ejemplo de solapamiento cuando x e y son iguales pero que no trae consecuencias en el resultado del algoritmo.

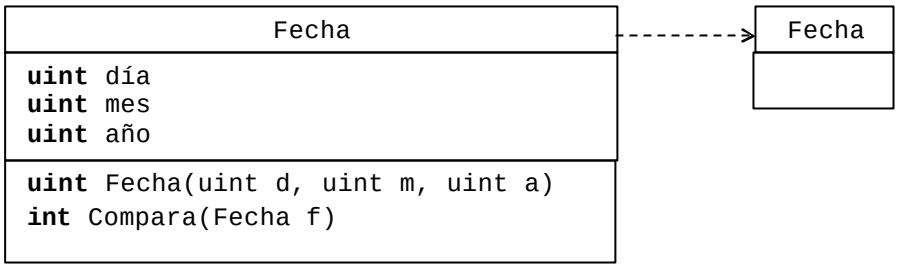
II.1.2.1 Caso de estudio: Refinamiento de la clase Fecha

Pregunta: ¿Cómo podríamos determinar entre dos personas la que nació primero?

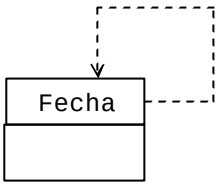
Nótese que con la comparación de las edades no es suficiente porque hasta el momento la edad se está calculando aproximadamente en años, por lo que cuando dos personas tienen la misma edad entonces no podemos asegurar nada. Es por ello que la comparación sería entre las fechas de nacimiento.

Evidentemente esta comparación entre las fechas de nacimiento no es la única comparación "posible" entre las instancias de la clase `Fecha`. Es decir, múltiples situaciones nos podrían llevar a este tipo de comparación (ejemplo, entre dos efemérides). Esto nos lleva a definir una *autorrelación* de uso o dependencia en la propia clase `Fecha`. Esta autorrelación será expresada en un método `Compara` que utiliza otra instancia de `Fecha` para realizar la mencionada comparación. Esta operación se establece retornando el valor 1 si el objeto

`Fecha` "dependiente" es menor que el objeto `Fecha` "del que se depende", 0 si son iguales y -1 en otro caso. En la figura siguiente se muestra un diagrama de clase donde se incorpora el referido método a través del cual se establece la autorrelación de uso `Fecha`.



Esta relación ha sido expresada de esta forma para intentar mayor legibilidad en estos inicios aunque también podría se expresada como se muestra en la siguiente figura.



Otra forma muy clara de ver este tipo de relación y las relaciones de dependencia o uso en general es con la presencia de algún parámetro de la clase "de la que se depende" en la clase "dependiente". En lo adelante apenas se utiliza esta notación para no sobrecargar el texto con recursos gráficos. De hecho las relaciones de dependencia o uso son las que más se usan a lo largo del texto.

Una variante de algoritmo para determinar que una `Fecha` es mayor que otra (retornar el valor 1) es que el año de la primera ("objeto de la clase dependiente") sea mayor que el año de la segunda ("objeto de la clase que se depende"), si son iguales los años entonces que el mes sea mayor y por último si el año y el mes son iguales que el día entonces sea mayor; de no cumplirse ninguna de las condiciones se verifica entonces si los tres valores son iguales luego las fechas son iguales y se retorna valor 0 y en el otro caso la primera es menor que la segunda y entonces se retorna el valor -1. En el siguiente se muestra una variante de codificación de este algoritmo.

```
public class Fecha
{
      public uint día, mes, año;
      public Fecha(uint d, uint m, uint a)
      {
            día = d;
            mes = m;
            año = a;
      }
      public int Compara(Fecha f)
      {
            if ( (año > f.año) ||
                 (año == f.año && mes > f.mes) ||
                 (año == f.año && mes == f.mes && día > f.día))
                return 1;
            else
                  if (año == f.año && mes == f.mes && día == f.día)
                        return 0;
                  else
                        return -1;
      }
}
```

Pregunta: ¿Qué diferencia existe entre los algoritmos implementados en I.2 y I.3 y los del presente capítulo?

Es necesario recordar en primer lugar que los algoritmos implementados en I.2 y I.3 se basaban en la descomposición a través de la secuenciación o descomposición secuencial pues se descomponían en una serie de acciones que se ejecutaban de modo secuencial. Sin embargo, en los algoritmos implementados en el presente capítulo, dependiendo de la evaluación de una expresión lógica se toman distintos caminos; esta es una de las aristas del *análisis de casos* como otro ejemplo de descomposición en la construcción de algoritmos.

Para Molina et al [9], el análisis de casos es una forma de descomposición algorítmica encaminada a disminuir la complejidad de un problema, dividiéndolo en subproblemas más pequeños que es posible resolver independientemente unos de otros. Los casos se pueden obtener, bien a través del análisis del dominio de los datos o bien mediante deducción a partir del resultado esperado. En el análisis de casos, los subproblemas se identifican particionando el dominio de los datos en varios subdominios o casos. Cada caso o subproblema es una restricción del problema inicial a uno de sus subdominios. La combinación de los subproblemas se realiza de forma condicional, de tal manera, que los datos iniciales indicarán qué subproblema es necesario resolver para obtener el resultado final.

Ejemplo: Determinar los subdominios presentes en el último listado

Primeramente se tiene que recordar que los subdominios se expresan mediante expresiones lógicas, es por ello que en el referido listado aparecen los siguientes subdominios:

a.
```
a.año > f.año || año == f.año && mes > f.mes ||
año == f.año && mes == f.mes && día > f.día
```

b.
```
año == f.año && mes == f.mes && día == f.día
```

c.
```
a.año < f.año || año == f.año && mes < f.mes ||
año == f.año && mes == f.mes && día < f.día
```

Es probable que a usted le sorprenda la presencia de la expresión que se relaciona en c. pues esta no está de forma explícita (y por o tanto no tiene que evaluarse), puede comprobar que esta expresión es la relacionada con la acción o instrucción **`return -1`**; dicha expresión no tiene que evaluarse explícitamente porque gracias a la cláusula **`else`** si ya fueron evaluadas las dos anteriores y resultaron falsas, entonces sin dudas esta tercera tiene que ser verdadera.

Situación de análisis

Teniendo en cuenta lo expuesto con anterioridad, es probable que la implementación del método `Compara` ya expuesto sea equivalente a la que se muestra a continuación:

```
public class Fecha
{
      public int Compara(Fecha f)
      {
            if (año > f.año ||
                año == f.año && mes > f.mes ||
                año == f.año && mes == f.mes && día > f.día)
               return 1;
            else
                if (año == f.año && mes == f.mes && día == f.día)
                      return 0;
                else
                      if (año < f.año ||
                          año == f.año && mes < f.mes ||
                          año == f.año && mes == f.mes && día < f.día)
                      return -1;
      }
}
```

Pregunta: ¿Cuál será el resultado de la compilación del código del listado anterior desde el punto de vista sintáctico y semántico?

Desde el punto de vista sintáctico todo está correcto porque todas las instrucciones están bien escritas. Sin embargo, desde el punto de vista semántico existe un problema: note que en este caso el retorno del valor por parte del método depende de la evaluación explícita de tres expresiones lógicas y como el compilador no tiene la garantía de que siempre una de las tres resulte verdadera (*tampoco es de su responsabilidad verificarlo*) entonces le informa un error semejante al siguiente: `no todas las rutinas de código devuelven un valor`. De todas formas esto no constituye un problema (además, sin dudas que el compilador nos está protegiendo), pues en nuestra opinión un código como el presentado en el listado anterior constituye una mala práctica de programación ya que es menos legible y más confuso que el presentado originalmente.

Situación de análisis

Note que en esta última variante en las subexpresiones se han ignorado los paréntesis del código original.

Pregunta: ¿Serán equivalentes entonces las expresiones?

Claro que si y el fundamento de esta respuesta es posible de encontrar en la prioridad de los operadores presentada con anterioridad en I.2.

Situación de análisis

Se desea disponer en la clase `Fecha` de una responsabilidad que les permita a sus objetos incrementarse en un día. Sin dudas esta responsabilidad sería útil para obtener la `Fecha mañana` a partir de un objeto `Fecha hoy`.

Este algoritmo sería muy simple, siempre y cuando el objeto `Fecha` en cuestión no se encuentre en el último día de su respectivo mes pues en ese caso solamente con incrementar el `día` resolvemos. Determinar si el mencionado objeto `Fecha` se encuentra en el último día de su respectivo mes nos lleva a una acción intermedia que es determinar los días del mes.

Pregunta: ¿Cuál sería el algoritmo para determinar los días del mes?

Un algoritmo para retornar los días de un mes se resume a devolver 30 en el caso de los meses 4, 6, 9, 11; cuando el mes sea 2 se devuelve 28 si el año no es bisiesto y 29 en caso contrario; finalmente se retorna 31 para el resto de los meses. Se observa en sus variantes de implementación que este es un ejemplo clásico para aplicar la mayoría de los elementos relacionados con el análisis de casos. Una variante de implementación de este algoritmo se presenta en el listado siguiente:

```
class Fecha
{
        ...
    public int  DíasMes()
    {
        if (mes == 4 || mes == 6 || mes == 9 || mes == 11)
            return 30;
        else
            if (mes == 2)
                if (bisiesto( ))
                    return 29;
                else
                    return 28;
            else
                return 31;
        }
        public bool bisiesto() {...}

}
```

Nótese como elemento significativo en este listado que en el caso de que el valor de un mes sea 2, entonces luego de verificarse la condición `mes == 2` en la instrucción **`if`**, se tiene que verificar otra condición, en este caso si el año es bisiesto por lo que entonces se tiene la evaluación de una nueva condición. Esto es lo que se conoce como **`if`** anidado, es decir un **`if`** como parte de otro **`if`**. Esto muestra la capacidad de la instrucción alternativa **`if`** para expresar el caso de que una sentencia que sigue a un **`if`** sea a su vez otro **`if`**, o que la sentencia que sigue a un **`else`** sea otro **`if`**, y esta circunstancia puede repetirse varias veces de forma que se disponga de una serie de decisiones anidadas. En caso de duda, lo mejor es delimitar entre llaves cada una de los bloques de código asociados a cada **`if`** o **`else`**, aunque el sangrado (indentado) propuesto como estilo de código en I.2.7 puede ser suficiente.

Ejemplo: Diseñar los métodos adecuados para obtener una aplicación que simule la problemática planteada en la última situación de análisis.

Una vez que se dispone del método `DíasMes` se puede implementar entonces el método `IncrementaDía`. El algoritmo en este caso consiste en determinar si el día de la `Fecha` en cuestión coincide con el último día de su respectivo mes (método `DíasMes`), en caso negativo apenas se incrementa el día en 1 y en caso contrario `día` toma valor 1 y el mes también se incrementa en 1 exceptuando cuando es 12 (diciembre), que entonces toma valor 1 (enero) y `año` se incrementa en 1. Posteriormente el método `Mañana` se implementa creando otra instancia de `Fecha(mañana)` a partir de las propios valores `día`, `mes` y `año`, luego se le pasa un mensaje a `mañana` para que se incremente un día y finalmente se retorna el objeto `mañana`. La implementación de estos métodos se puede encontrar en el listado siguiente.

```
class Fecha
{
      public void IncrementaDía()
      {
            if (día != DíasMes())
                  día++;
            else
            {
                  día = 1;
                  if (mes != 12)
                        mes++;
                  else
                  {
                        mes = 1;
                        año++;
                  }
            }
      }
      public Fecha Mañana()
      {
            Fecha mañana = new Fecha(día, mes, año);
            mañana.IncrementaDía();
            return mañana;
      }
}
```

Al analizar la implementación del algoritmo del método `IncrementaDía`, es de notar que existen dos casos mas generales que se definen a través de las expresiones de los subdominios:

`día != DíasMes()` y `día == DíasMes()`. El subproblema asociado con el segundo caso a su vez se descompone en una acción de asignación y en otro análisis de casos (ahora con el mes). Es decir, se descompone en dos acciones que se ejecutan secuencialmente (composición secuencial) pero a su vez una de estas (la segunda) se resuelve a través de un análisis de casos. Este es un ejemplo de como la descomposición de un problema en subproblemas también se puede obtener a partir de la combinación de la composición y el análisis de casos.

Por último, la aplicación se obtiene apenas creando una instancia de `Fecha` (`hoy`) a partir de la fecha actual (la podemos obtener a través de la clase `DateTime` de C#) y luego se le pasa un mensaje al objeto hoy para que nos retorne la fecha de mañana (método `Mañana`).

```
{
      uint d = (uint)DateTime.Now.Day;
      uint m = (uint)DateTime.Now.Month;
      uint a = (uint)DateTime.Now.Year;
      Fecha hoy = new Fecha(d, m, a);
      Fecha mañana = hoy.Mañana();
}
```

Pregunta: ¿Son imprescindibles los *casting* que se realizan en el listado anterior?

Para responder esta pregunta se debe comenzar por recordar que los parámetros del constructor de la clase `Fecha` son de tipo **`uint`**, es por ello que las variables `d`, `m`, y `a` también se han definido como **`uint`** pues en su momento se pasarán como parámetro al constructor de `Fecha`. Sin embargo, `Day`, `Month`, `Year` (aun no es posible precisar que recurso del lenguaje son) retornan valores **`uint`**; es decir si no se realiza el *casting* entonces el compilador retorna errores con el mensaje: `no se puede convertir implícitamente del tipo 'int' a 'uint'`. Estas son las razones que justifican y hacen imprescindible para este caso el uso del *casting*.

Nota: Las dos instrucciones siguientes son equivalentes al último listado presentado:

```
      hoy = new Fecha((uint)DateTime.Now.Day,
                      (uint)DateTime.Now.Month,
                      (uint)DateTime.Now.Year);
      mañana = hoy.Mañana();
```

II.1.3 Estructura de control alternativa múltiple (switch case)

En la implementación del método `DíasMes` se puede apreciar cómo para diferentes valores se toman distintas determinaciones, en casos donde sean muchos los valores resulta poco legible el uso de los **`if`** anidados como se muestra en el siguiente ejemplo.

Ejemplo: Implementar una variante del método `DíasMes` de la sección anterior donde se invierta el orden de evaluación de los dos últimos casos subdominios más generales.

```
  public int  DíasMes()
  {
      if (mes == 4 || mes == 6 || mes == 9 || mes == 11)
            return 30;
      else
            if (mes == 1 || mes == 3 || mes == 5 || mes == 7 ||
                mes == 8 || mes == 10 || mes == 12)
                  return 31;
            else
                  if (bisiesto( ))
                        return 29;
                  else
                        return 28;
  }
```

Con el objetivo de hacer más legibles y claros los códigos de implementación de los algoritmos donde se tienen múltiples casos, la mayoría de los lenguajes brinda una *instrucción de alternativa múltiple* (**`switch case`**) cuya sintaxis se muestra seguidamente:

```
      switch (<expresión>)
      {
            case <valor 1>: <bloque de instrucciones 1>
                            [break]
```

```
            case <valor 2>: <bloque de instrucciones 2>
                            [break]

            case <valor n>: <bloque de instrucciones n>
                            [break]
            [default: <bloque de instrucciones default>
                      [break]]
      }
```

La semántica de esta instrucción es muy simple: si al evaluarse `<expresión>` toma `<valor 1>` entonces se ejecuta `<bloque de instrucciones 1>`; si al evaluarse `<expresión>` toma `<valor 2>` entonces se ejecuta `<bloque de instrucciones 2>` y así sucesivamente hasta `<valor n>`. De no coincidir la evaluación con alguno de las valores predeterminados y existir la rama **`default`** (note que es opcional), entonces se ejecutará `<bloque de instrucciones default>`. La instrucción **`break`** es obligatoria excepto cuando exista una instrucción **`return`** en el respectivo bloque de instrucciones. Este **`break`** indica que después de ejecutar el bloque de instrucciones que lo precede se salta a la próxima instrucción después del **`switch`**.

Nota: Existen otros recursos que se pueden utilizar y con los que es posible obtener resultados similares a **`break`** pero en el presente texto no se presentan pues constituyen una mala práctica de programación

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.

Para ejemplificar esta nueva estructura sintáctica se define a continuación una variante del método `DíasMes`

```
public int  DíasMes()
{
      switch (mes)
      {
            case 4: return 30;
            case 6: return 30;
            case 9: return 30;
            case 11: return 30;
            case 2 : if (bisiesto( ))
                        return 29;
                     else
                        return 28;
            default: return 30;
      }
}
```

En el caso que varias ramas coincidan en sus respectivos bloques de instrucciones, éstas se pueden agrupar como se muestra seguidamente:

```
public int  DíasMes()
{
      switch (mes)
      {
            case 4: case 6: case 9: case 11: return 30;
            case 2 : if (bisiesto( ))
                        return 29;
                     else
                        return 28;
            default: return 30;
}
```

A continuación se muestra una variante del método `Main` de la aplicación `Hola Mundo` *personalizada*, (idea original en [12]).

```
public static void Main(string[] args)
{
      if (args.Length > 0)
            switch(args[0])
            {
                  case "José":
                       Console.WriteLine("Hola José. Buenas");
                       break;
                  case "Paco":
                       Console.WriteLine("Hola Paco. " +
                                         "Me alegro de verte");
                       break;
                  default:
                       Console.WriteLine("Hola {0}", args[0]);
                       break;
            }
      else
            Console.WriteLine("Hola Mundo");
}
```

Esta aplicación reconoce ciertos nombres de personas que se le pueden pasar como argumentos al ejecutarlo y les saluda de forma especial (`"José"` y `"Paco"`). La rama **`default`** se incluye para dar un saludo por defecto a las personas no reconocidas. Note en este segmento de código la combinación de una alternativa simple con una múltiple.

Nota: En esta aplicación se reconoce la presencia de elementos nuevos (`Main(string[] args)`, `args[0]`, etc.), que no se aclaran hasta III.1 donde será capaz de explicarlos por si solo.

II.1.4 Caso de estudio: clase `Triángulo`

Ejemplo: Definir una clase `Triángulo` a partir de la longitud de sus lados. Implementar un método `Clasifica` que permita obtener la clasificación de las respectivas instancias en cuanto a las respectivas longitudes de los lados (equilátero, isósceles y escaleno).

Para implementar el algoritmo que determina la clasificación del triángulo es evidente que se tiene un análisis de casos con tres subdominios: los tres lados iguales (equilátero), solamente dos lados iguales (isósceles) y los tres lados diferentes (escaleno). El diseño e implementación de esta clase se explican por si solos en el siguiente listado.

```
public class Triángulo
{
    public uint lado1, lado2, lado3;
    public Triángulo(uint lado1, uint lado2, uint lado3)
    {
          this.lado1 = lado1;
          this.lado2 = lado2;
          this.lado3 = lado3;
    }
    public string Clasifica()
    {
          if (lado1 == lado2 && lado2 == lado3)
                return "Equilátero";
          else
                if (lado1 == lado2 || lado2 == lado3 || lado3 == lado1)
                      return "Isósceles";
                else
                      return "Escaleno";
```

```
    }
}
```

Situación de análisis

Sin dudas, algunos de los lectores pensaron en la expresión

`(lado1 == lado2 == lado3)`, en vez de

`(lado1 == lado2 && lado2 == lado3)`.

Pregunta: ¿Cuál será la "reacción" del compilador de C# ante la presencia de la primera de las expresiones mostradas con anterioridad?

En esta situación el compilador retorna un mensaje de error similar al siguiente: El operador '`==`' no se puede aplicar a operandos '**`bool`**' y '**`int`**'. Esto esta dado porque se presenta una expresión compuesta por operadores de igual precedencia, en este caso se evalúa primeramente la subexpresión de la izquierda (`lado1 == lado2`) que devuelve un valor **`bool`** que tiene que compararse ahora con otro que es **`int`**, lo cual no es posible desde el punto de vista semántico. Teniendo en cuenta estos elementos se puede asegurar que la siguiente expresión es válida en C#:

```
1 == 1 == true
```

Ejemplo: Refine el algoritmo del método `Clasifica` teniendo en cuenta que los lados del triángulo se mantienen ordenados de menor a mayor (idea original en [8]).

En este caso ha cambiado el estado inicial del algoritmo y esto implica también cambios en los subdominios como se aprecia en el siguiente listado.

```
public string Clasifica()
{
    if (lado1 == lado3)
          return "Equilátero";
    else
          if (lado1 == lado2 || lado2 == lado3)
                return "Isósceles";
          else
                return "Escaleno";
    }
}
```

II.1.5 Caso de estudio: aplicación `Imprimir calificación final`

Ejemplo: Escribir una aplicación donde se imprima la calificación final de una asignatura (*suspenso* (menos de 14), *aprobado* (14..15), *notable* (16..17), *sobresaliente* (18..19), *matrícula de honor* (20)) a partir de las preguntas contestadas correctamente de un examen tipo test de 20 preguntas (idea original en [9]).

No cabe la menor duda que este es un clásico ejemplo de alternativa múltiple como se muestra en el listado siguiente:

```
uint nota = uint.Parse(Console.ReadLine());

switch (nota)
{
    case 20: Console.WriteLine("M. Honor");
        break;
    case 19: case 18: Console.WriteLine("Sobresaliente");
        break;
    case 17: case 16: Console.WriteLine("Notable");
        break;
    case 15: case 14: Console.WriteLine("Aprobado");
```

```
        break;
    default: Console.WriteLine("Suspenso");
        break;
}
```

Pregunta: ¿Qué sucede si no se define rama `default` del listado anterior?

Cuando la nota es menor que 14 no se imprime ningún mensaje. Una vez más se pone de manifiesto el típico error ya mencionado, *olvido de casos*. Sin embargo no sucede lo mismo con el *solapamiento* porque no es posible repetir un valor en dos ramas, esta es realmente una fortaleza de la estructura alternativa múltiple de C#.

Pregunta: Imagínese ahora que no se olvida ningún caso y la rama `default` se define para el caso *aprobado*. ¿Cómo se trataría el caso *suspenso*?

En este caso quedaría una rama con 14 casos para considerar todos los casos de *suspenso* como se muestra a continuación:

```
case 13: case 12: case 11: case 10: case 9: case 8: case 7:
case  6: case  5: case  4: case  3: case 2: case 1: case 0:
     Console.WriteLine("Suspenso");
     break;
```

Nota: Sin dudas que esto hace concluir que se debe ser muy cuidadoso al elegir el orden de los casos a la hora de diseñar las diferentes ramas.

II.1.6 Caso de estudio: clase `Calculadora`

Ejemplo: Diseñar e implementar una clase `Calculadora` de números enteros con un método `Calcula` a través del cual se implementen las cuatro operaciones aritméticas básicas: suma, resta, multiplicación y división entera. Los datos de entrada para el método `Calcula` serán los dos operados seguidos del operador, este último representado por un valor de tipo `char` ('+', '-', '*' y '/' respectivamente).

Por lo simple del diseño de la clase y del algoritmo del método `Calcula`, seguidamente se presenta la implementación de la clase que se explica por si sola en el listado II.1.4.4.1.

```
public class Calculadora
{
    public int Calcula(int operando1, int operando2, char operador)
    {
          switch (operador)
          {
                case '+': return operando1 + operando2;
                case '-': return operando1 - operando2;
                case '*': return operando1 * operando2;
                default: return operando1 / operando2;
          }
    }
}
```

II.1.7 Caso de estudio: clase `EficaciaBateria`

Situación de análisis (idea original en [9])

El ejército ha calculado la eficacia de batir un objetivo terrestre por una batería en función de tres elementos: Disponer de boletín meteorológico (B), disponer de observador (A), llevar más de 30 minutos en posición (T).

Ejemplo: Implementar una clase que permita determinar la eficacia de batir un objetivo terrestre por una batería a partir de la siguiente tabla:

A	B	T>30	Grado Eficacia
no	no	si	0.3
no	no	no	0.1
no	si	si	0.7
no	si	no	0.4
si	no	no	0.2
si	no	si	0.6
si	si	si	0.8
si	si	no	0.5

Sin dudas que lo más interesante de la clase propuesta es el método para determinar la eficacia que se implementa a partir de la determinación de los respectivos subdominios explícitos en la tabla anterior.

En un primer refinamiento es posible obtener la implementación que se presenta a continuación:

```
public class EficaciaBatería
{
      bool obs;   // ¿Tiene observador?
      bool bM;    // ¿Dispone de boletín meteorológico?
      uint t;     // Tiempo en posición
      public EficaciaBatería(bool obs, bool bM, uint t)
      {
            this.obs = obs;
            this.bM = bM;
            this.t = t;
      }
      public double Eficacia()
      {
            if (!obs && !bM && !(t <= 30))
                return 0.3;
            else
                if (!obs && !bM && (t <= 30))
                    return 0.1;
                else
                    if (!obs && bM && !(t <= 30))
                        return 0.7;
                    else
                        if (!obs && bM && (t <= 30))
                            return 0.4;
                        else
                            if (obs && !bM && (t <= 30))
                                return 0.2;
                            else
                                if (obs && !bM && !(t <= 30))
                                    return 0.6;
                                else
                                   if (obs && bM && !(t <= 30))
                                       return 0.8;
                                   else
                                       return 0.5;
      }
}
```

Es evidente que a la solución anterior le resta claridad y legibilidad en número de anidamientos entre los **`if`**. A continuación se presenta una solución interesante y que puede servir de esquema en la solución de futuros problemas con características similares.

Si se le asigna un valor entero a cada condición (`A (2)`, `B (3)`, `T>30 (4)`) de forma tal que el `acumulado` de sumar dicho valor (en caso de cumplirse la respectiva condición) ó 0 (cero) en caso de que no se cumpla la condición sea diferente para todas las filas de la tabla, entonces la solución sería tan simple como hacer una estructura **`switch case`** para el `acumulado`.

Seguidamente se muestra las modificaciones a la tabla de los grados de eficacia incorporando los valores y los acumulados:

A (2)	**B** (3)	**T>30** (4)	**Acumulado**	**Grado Eficacia**
no	no	si	4	0.3
no	no	no	0	0.1
no	si	si	7	0.7
no	si	no	3	0.4
si	no	no	2	0.2
si	no	si	6	0.6
si	si	si	9	0.8
si	si	no	5	0.5

A continuación se presenta una variante de implementación de esta propuesta de solución:

```
public double Eficacia1()
{
    int acumulado = 0;
    if (obs)
        acumulado += 2;
    if (bM)
        acumulado += 3;
    if (t > 30)
        acumulado += 4;
    switch (acumulado)
    {
          case 4: return 0.3;
          case 0: return 0.1;
          case 7: return 0.7;
          case 3: return 0.4;
          case 2: return 0.2;
          case 6: return 0.6;
          case 9: return 0.8;
          default: return 0.5;
    }
}
```

Finalmente y aprovechando alguna relación existente entre los valores del acumulado y la eficacia puede obtenerse el refinamiento que se muestra a continuación:

```
public double Eficacia2()
{
      int acumulado = 0;
      if (obs)
          acumulado += 2;
      if (bM)
          acumulado += 3;
      if (t > 30)
          acumulado += 4;
```

```
        switch (acumulado)
        {
            case 4:
            case 9: return acumulado / 10.0 - 0.1;
            case 0:
            case 3: return acumulado / 10.0 + 0.1;
            default: return acumulado / 10.0;
        }
}
```

II.1.8 Ejercicios

1. Diseñe e implemente una clase `Estudiante` que almacene las notas de 3 asignaturas y permita obtener su respectivo promedio.

2. Auxiliándose de la clase `SecretariaDocente` refinada en el capitulo actual, obtenga una aplicación que permita determinar si un estudiante puede pertenecer al Cuadro de Honor de la facultad.

3. Obtenga una aplicación que permita obtener el mayor entre 3 números enteros.

4. Auxiliándose del desarrollo del caso de estudio presentado en la sección II.1.2.1 escriba una aplicación que responda a las siguientes orientaciones:

- Crear dos instancias de la clase `Fecha` (`f1` y `f2`).
- Determine la `Fecha` menor entre `f1` y `f2` (`fechaMenor`).
- Obtenga la fecha siguiente a `fechaMenor` y determine la relación que existe con respecto a la otra `Fecha`.

5. Obtenga una aplicación que permita crear una instancia de la clase `Triángulo` definida en la solución al ejercicio II.1.4.1 e imprima su respectiva clasificación.

6. Refine la clase `Triángulo` incorporándole un método que nos permita ordenar los lados para entonces utilizar la implementación del método `Clasifica` que se muestra en II.1.4.

7. Utilizando la clase `Calculadora` que se muestra en II.1.6 obtenga una aplicación que lea dos números enteros e imprima su respectiva suma, resta, producto y división.

8. La nota final de un estudiante se calcula a partir de las notas de los tres exámenes parciales, calculando la media de las dos mejores notas. Obtenga una aplicación que permita leer el nombre del estudiante y las tres notas de los parciales e imprima la nota final. Para ello auxíliese del diseño e implementación de una clase `Estudiante`.

9. Refine el algoritmo de la clase `DesgloseDinero` (seccion I.2.10) de forma tal que no se realicen operaciones innecesariamente. Es decir, si la cantidad de dinero a desglosar es 100 pesos, no hay porqué calcular los respectivos desgloses de 50, 20, etc. pues de antemano sabemos que son cero.

10. Defina una clase (`Marcador`) que simule la evolución del marcador en un partido de tenis. Cada vez que se ha jugado una bola el árbitro indica que jugador ha ganado. Las instancias de tipo `Marcador` deben poder mostrar en cualquier momento el marcador del partido.

11. Defina una clase que permita representar polinomios de segundo grado $P(x) = ax^2 + bx + c$. Permita obtener las raíces de estos polinomios, es decir, los valores de x tal que $P(x) = 0$. Obtenga una aplicación consola que imprima las raíces de los siguientes polinomios:

1. $P(x) = x^2 - 4$ y $P(x) = x - 4$.

12. Refine la clase para representas puntos de R^2 de forma tal que se permita determinar en que cuadrante se encuentra un determinado punto. Para los puntos que están sobre los ejes de coordenadas se establece el siguiente convenio comenzando por los que se encuentran en la parte positiva del eje x: -1, -2, -3, -4 y 0 para el (0, 0).

13. Refine la clase para representar vectores de forma tal que nos permita determinar si dos vectores son paralelos u ortogonales.

14. Refine la clase matriz cuadrada de orden 2 de forma tal que nos permita determinar si estamos en presencia de una matriz nula, identidad, etc.

15. Tomando la clase Triángulo implementada en II.1.4, permita determinar si:

a) Es acutángulo, rectángulo, obtusángulo.

b) Es semejante o igual a otro triángulo dado.

16. Incluya en la clase Circunferencia diseñada en el capítulo I.2 métodos que permitan:

c) Determinar en que cuadrante esta situada la circunferencia dado las coordenadas de su centro.

d) Conocer si la misma está contenida totalmente en ese cuadrante.

17. Incluya en el diseño de la clase Estudiante realizado en el ejercicio 8 un método que permita calificar la nota final según el criterio siguiente (tome que las notas parciales son números reales entre 0-100).

`100 >= nota >=95`	`Excelente`
`95 > nota >=85`	`Muy bien`
`85 > nota >=75`	`Bien`
`75 > nota >=65`	`Regular`
`65 > nota`	`Mal`

18. Obtenga una aplicación que dada la edad de una persona, permita saber (consultando el reloj de la máquina) si es su cumpleaños e indique que edad cumple. Utilice la clase Persona implementada en I.2.

19. Diseñe e implemente una clase Trabajador que conociendo de cada trabajador su nombre, fecha de nacimiento, salario básico, área donde trabaja y la cantidad de días trabajados en la quincena, permita entre otras responsabilidades:

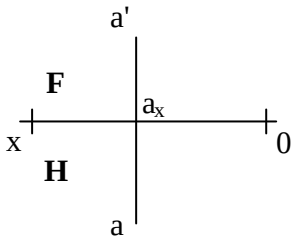
a) Conocer el salario que deberá cobrar dicho trabajador.

b) Si es vanguardia.

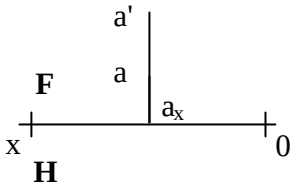
c) Si esta próximo a la jubilación.

Notas: Un trabajador es vanguardia si ha trabajado más del 87% de los días de la quincena. Un trabajador deberá jubilarse a los 60 años de edad.

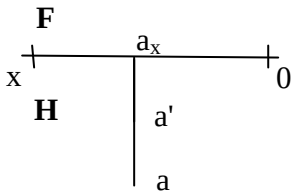
20. Una tarea de gran importancia dentro de la Geometría descriptiva es saber en que cuadrante se encuentra ubicado un punto a partir de sus proyecciones en un sistema ortogonal. Elabore una clase que permita almacenar la información necesaria para poder representar puntos que se encuentran en un sistema ortogonal (ver Figura No.1). Los puntos serán introducidos a partir de sus coordenadas x,y,z. Considere que los puntos están siempre en el primer cuadrante y sólo se representaran el plano Frontal y Horizontal. En la tabla No.1 se muestra un resumen de cómo quedarían los signos en cada cuadrante.



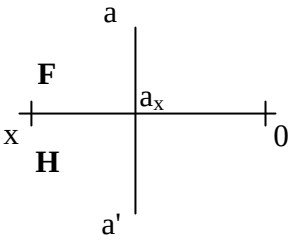
a) Punto en el primer cuadrante



b) Punto en el segundor cuadrante



c) Punto en el tercer cuadrante



d) Punto en el Cuarto cuadrante

Figura No.1. Puntos representados en abatimiento en los cuatro cuadrantes.

Tabla No.1. Signos de las coordenadas en los diferentes cuadrantes.

	I	II	III	IV
x	+	+	+	+
y	+	-	-	+
z	++	+	-	-

21. Según la posición de un segmento de recta con respecto a los planos de proyección Frontal, Horizontal y Lateral (F, H, L), se clasifican en:

Segmentos de recta paralelos a los planos.

a) Recta frontal (paralela al plano frontal)
b) Recta horizontal (paralela al plano horizontal)
c) Recta lateral (paralela al plano lateral)

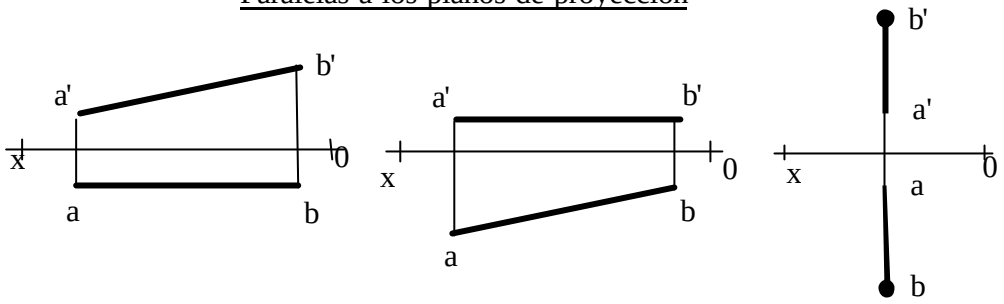
Segmentos de recta perpendiculares a los planos.

a) Proyectante frontal (perpendicular al plano frontal)
b) Proyectante horizontal (perpendicular al plano horizontal)
c) Proyectante lateral (perpendicular al plano lateral)

Segmentos de recta ni paralelos, ni perpendiculares.

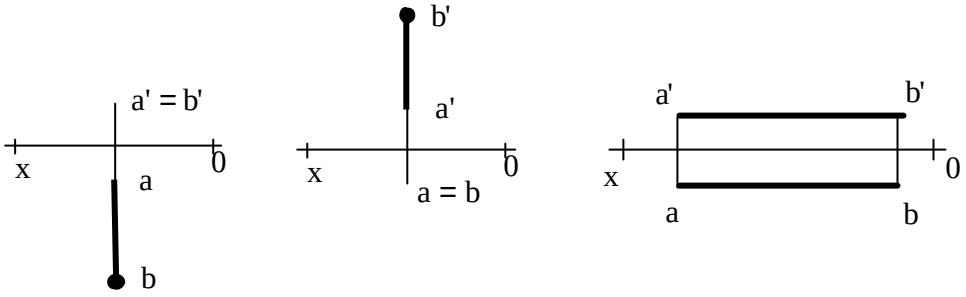
a) Recta oblicua (no es ni paralela, ni perpendicular con respecto a los planos de proyección).

Paralelas a los planos de proyección

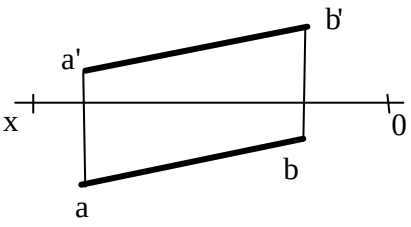


a) Frontal b) Horizontal c) Lateral

Perpendiculares a los planos de proyección



a) Proyectante frontal b) Proyectante horizontal c) Proyectante lateral



a) Oblicua

Basado en la explicación anterior, elabore una clase que permita clasificar los segmentos de recta según su posición respecto a los planos de proyección. Los segmentos de recta se introducen mediante las coordenadas de los puntos extremos A(x, y, z) y B(x, y, z).

II.1.9 Bibliografía complementaria

- Capítulos 2 de Jesús García Molina et al., Introducción a la Programación, Diego Marín ,1999.
- Capítulo 16 de Jose A. González Seco, El lenguaje de programación C#, 2002. Puede descargarse en http://www.josanguapo.com/

Capítulo II.2

II.2 Componentes objetos

Desde el punto de vista sintáctico, las clases encapsulan responsabilidades de los futuros objetos y como se ha visto con anterioridad éstas definen estructura y comportamiento de los mismos. Desde el punto de vista semántico las clases definen un tipo de dato abstracto.

Hasta el momento se han implementado (codificado) las clases prácticamente de forma aislada, salvo algunas relaciones de uso que se han definido solamente a través de parámetros o valor de retorno en los métodos. Es decir, en los capítulos precedentes se han implementado clases donde sus variables solamente son de tipos básicos del lenguaje. Sin embargo, como se presentó en el capítulo I.1 las variables a su vez pueden ser objetos.

II.2.1 Componentes objetos

Pregunta: ¿Cómo podríamos implementar en C# la relación de uso que se establece entre `Persona` y `Fecha` presentada en el capítulo anterior?

El elemento más importante aquí es tener en cuenta que algunas instancias de la clase `Fecha` pasan a formar parte de la clase `Persona` a través de la variable `fechaNacimiento`. Es decir, en primera instancia se tiene que definir una variable `Fecha fechaNacimiento` en la clase `Persona` y luego un parámetro también de tipo `Fecha` en el constructor de la clase `Persona` para garantizar la inicialización de la referida variable `fechaNacimiento`. Esta inicializacion por parte del constructor se va a lograr a través del operador **`new`** o asignando la referencia a un objeto ya creado con anterioridad. También el método `Edad` tiene que redefinirse en función de esta nueva variable, etc. Seguidamente se muestra una variante de implementación de la referida relación de uso entre `Persona` y `Fecha`.

```
public class Persona
{
    string nombre;
    string apellidos;
    Fecha fechaNacimiento;
    public Persona(string unNombre, string unAp, Fecha unaFecha)
    {
          nombre = unNombre;
          apellidos = unAp;
          fechaNacimiento = unaFecha;
    }
    public long Edad()
    {
          return System.DateTime.Now.Year - fechaNacimiento.año;
    }
}
```

Ejemplo: Utilizando la clase definida con anterioridad, obtenga una aplicación consola donde se defina una instancia de `Persona` y se imprima su edad.

Una consecuencia de que hasta el momento las clases definidas solamente disponen de variables de tipos de datos básicos es que los parámetros del constructor también han sido de estos tipos. Sin embargo, ahora disponemos de una variable que a su vez es un objeto (`fechaNacimiento`) y por lo tanto debe ser inicializada a través del constructor; por lo que una variante para crear las instancias de la clase `Persona` puede ser creando previamente la instancia que representa su fecha de nacimiento que luego será pasada como parámetro al referido constructor como se muestra seguidamente:

```
{
    Console.WriteLine("Datos de la persona");
    Console.Write("Nombre: ");
    string nombre = Console.ReadLine();
    Console.Write("Apellidos: ");
    string apellidos = Console.ReadLine();
    Console.WriteLine("Fecha de nacimiento: ");
    Console.Write("Día: ");
    uint d = uint.Parse(Console.ReadLine());
    Console.Write("Mes: ");
    uint m = uint.Parse(Console.ReadLine());
    Console.Write("Año: ");
    uint a = uint.Parse(Console.ReadLine());
    Fecha f = new Fecha(d, m, a);
    Persona unaPersona = new Persona(nombre, apellidos, f);
    Console.WriteLine("Su edad es: " + unaPersona.Edad().ToString());
    Console.WriteLine();
}
```

Situación de análisis

Analice el segmento de código que se muestra a continuación:

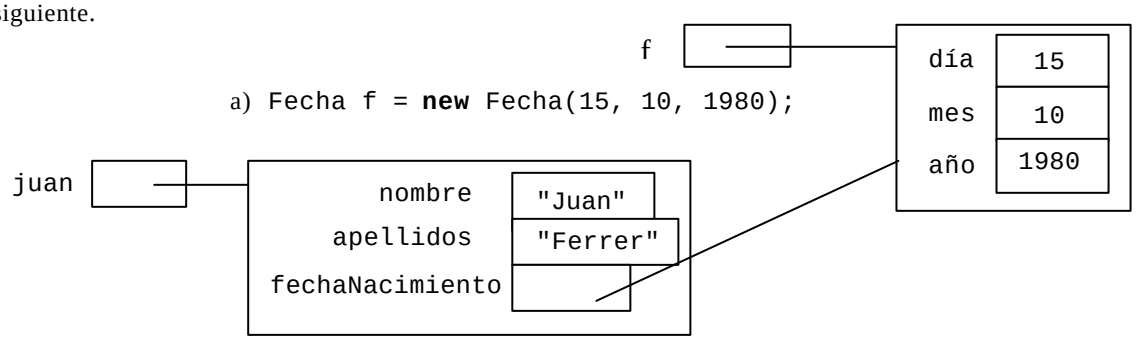
```
{
      Fecha f = new Fecha(15, 10, 1980);
      Persona juan = new Persona("Juan", "Ferrer", f);
      Console.WriteLine("{0} su edad aproximada es {1}",
                        juan.nombre,juan.Edad());

      f.año = 2000;
      Console.WriteLine("{0} su edad aproximada es {1}",
                        juan.nombre,juan.Edad());
}
```

Pregunta: ¿Qué valores se imprimen al ejecutar el código del listado con anterioridad?

Se imprimen los siguientes mensajes (asumiendo que el año actual es 2003):

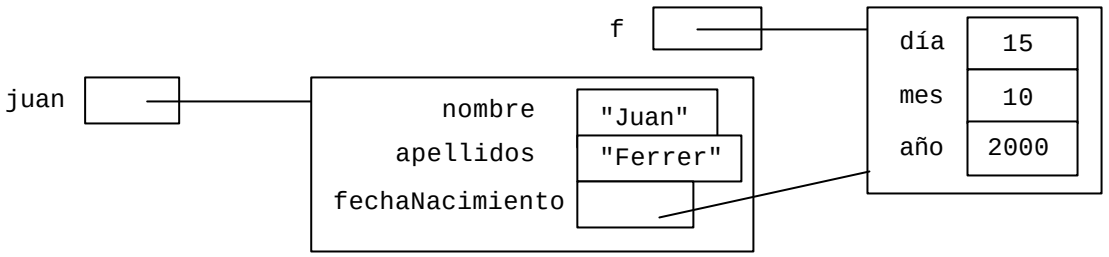
```
Juan su edad aproximada es 23
Juan su edad aproximada es 3
```

Esto se justifica por la semántica por referencia que utilizan los objetos. Es decir, la variable `f` y la variable `juan.fechaNacimiento` hacen referencia al mismo objeto por lo que cualquier cambio en `f` también se manifiesta en `juan.fechaNacimiento`. Ver la representación de este suceso en la figura siguiente.



a) `Fecha f = new Fecha(15, 10, 1980);`

b) `Persona juan = new Persona("Juan", "Ferrer", f);`



c) `f.año = 2000;`

Para solucionar este problema es posible crear el objeto `juan` evitando que ningún otro objeto haga referencia a su variable `fechaNacimiento` como se muestra a continuación:

```
Persona juan = new Persona("Juan", "Ferrer", new Fecha(15, 10, 1980));
```

Note que ahora se ha creado la fecha de nacimiento del objeto `juan` directamente en el constructor. Los objetos que se crean directamente en el llamado a un método reciben el nombre de *objetos anónimos*.

Pregunta: ¿Será posible la creación del objeto `juan` de la siguiente forma?

```
Persona juan = new Persona("Juan", "Ferrer", 15, 10, 1980);
```

Esto es perfectamente factible pero entonces se necesita de otro constructor que ahora se encargue de crear la variable `fechaNacimiento` a partir de los respectivos parámetros para las variables `día` (15), `mes` (10) y `año` (1980) como se muestra seguidamente:

```
public class Persona
{
      public Persona(string unNombre, string unAp, Fecha unaFecha) ...
      public Persona(string unNombre, string unAp,
                     uint d, uint m, uint a)
      {
            nombre = unNombre;
            apellidos = unAp;
            fechaNacimiento = new Fecha(d, m, a);
      }
}
```

Pregunta: ¿Cómo es posible justificar la existencia de dos constructores en la clase `Persona`?

La respuesta a esta pregunta es simple y se obtiene del capítulo I.1 cuando se presentaron las ideas relacionadas con los constructores, definiéndose éstos como casos particulares de métodos. Siendo entonces aplicable para estos el concepto de sobrecarga de métodos, siempre y cuando se sigan las reglas para el uso de este recurso, las cuales se cumplen en el ejemplo anterior (nótese las diferencias entre las listas de parámetros de ambos constructores)

En este ejemplo se evidencia la importancia que tiene la sobrecarga de métodos, ya que permite definir diferentes variantes del método constructor de la clase `Persona`, posibilitando con ello varias formas de crear un objeto en función de los datos que se tengan y además eliminar ciertas situaciones que como se presentó en el ejemplo pueden implicar errores difíciles de detectar.

Estos dos constructores posibilitan la creación de instancias de la clase `Persona` de diferentes formas, con el primero de ellos ya debe haberse creado con anterioridad el objeto `Fecha` correspondiente a la fecha de nacimiento de la `Persona`, mientras que en el segundo es responsabilidad del propio constructor la creación del componente objeto `fechaNacimiento`. Con estas dos variantes es factible crear tres objetos `Persona` con estados muy similares a través de la misma variable (`juan`) y de diferentes formas como se muestra a continuación:

```
{…
      Fecha fNacimiento = new Fecha(15, 10, 1980);

      Persona juan = new Persona("Juan" ,  "Ferrer" , fNacimiento);

      juan =  new Persona("Juan" ,  "Ferrer" , 15, 10, 1980);

      juan =  new Persona("Juan" ,  "Ferrer" , new Fecha(15, 10, 1980));
}
```

Situación de análisis

Analice los respectivos constructores que se han definido para la clase `Persona`. Note que en ambos casos prácticamente el código es el mismo. Cuando cosas como esta suceden, generalmente puede haber una forma de reutilizar código.

Pregunta: ¿Cómo es posible definir el segundo constructor de la clase `Persona` en función del primero?

Como la indicación es definir el segundo constructor en función del primero, lo que se debe hacer es determinar cuáles son los parámetros que necesita el primer constructor para poder crear las respectivas instancias. En este caso se tiene que este constructor necesita de dos valores de tipo **`string`** (un`Nombre` y un`Ap`) y un objeto de tipo `Fecha`. (`unaFecha`). En el caso del segundo constructor también se tienen dos cadenas de caracteres y una terna a través de la cual puede ser creado un objeto `Fecha`, es decir se pueden obtener precisamente los parámetros que necesita el primer constructor:

`unNombre, unAp` y **`new`** `Fecha(15, 10, 1980)`

C# permite reutilizar el código entre los constructores a través de la palabra reservada **`this`** como se presenta a continuación:

```
public class Persona
{
    public Persona(string unNombre, string unAp, Fecha unaFecha) ...
    public Persona(string unNombre, string unAp,
                   uint d, uint m, uint a)
          : this(unNombre, unAp, new Fecha(d, m, a)){}
}
```

En el listado anterior se aprecia un nuevo uso de la palabra reservada **`this`**, esta vez como inicializador ya que al utilizarse inmediatamente después de la signatura de un constructor significa que se está invocando a otro constructor de la clase cuya signatura coincide con los parámetros que se están pasando; nótese que el primer constructor de `Persona` necesita como parámetros de un **`string`** y un objeto `Fecha`, luego en este caso los parámetros formales `unNombre` y `unAp` del segundo constructor pasan a ser parámetros reales en la llamada al primer constructor, mientras que los parámetros formales `d`, `m`, `a` también del segundo constructor son utilizados como parámetros reales en la creación del objeto anónimo que también se trasfiere en la llamada al primer constructor.

Debe especificarse que este uso de **`this`** es conocido como *inicializador* y es importante señalar que la llamada a un constructor a través del inicializador **`this`** se hace antes de hacer cualquier instrucción del propio constructor en el que se le invoca. Además, el inicializador **`this`** solamente puede ser utilizado en el constructor, si va a realizarse una llamada a un constructor sobrecargado.

Pregunta: ¿Cómo se puede acceder a las responsabilidades de las instancias que forman parte de otras instancias? Es decir, ¿cómo se puede acceder al día de nacimiento del objeto `juan` después de creado por cualquiera de las variantes?

La respuesta a la pregunta anterior se obtiene por intuición, es decir, si se aplica un pensamiento recursivo (IV.1): se accede primero a la instancia del objeto (`objetoContenedor`) que contiene como uno de sus campos otra instancia de alguna clase (`objetoComponente`) y luego a través de `objetoContenedor` y utilizando el operador "`.`" se accede a `objetoComponente` y finalmente a

través de esta última instancia y nuevamente utilizando el operador "`.`" se accede a las respectivas responsabilidades como se muestra a continuación:

```
objetoContenedor.objetoComponente.Responsabilidad
```

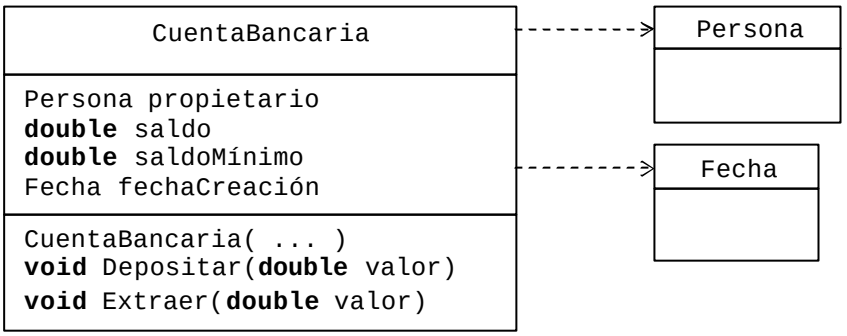
que en el caso particular del ejemplo entonces sería

```
juan.fechaNacimiento.día
```

II.2.1.1 Caso de estudio: Cuenta Bancaria

Ejemplo: Defina una clase que permita representar cuentas bancarias de un banco (`CuentaBancaria`) que se responsabilice con la información del propietario (`Persona`), la fecha de creación de cada cuenta, el saldo y el saldo mínimo que se debe tener para mantener la cuenta, que es previamente fijado por el banco y debe ser el mismo para todos los clientes. Además se debe permitir la realización de depósitos y extracciones.

Nuevamente se presenta una relación de uso, en este caso entre `CuentaBancaria` y `Persona` donde la primera define la responsabilidad propietario a través de la segunda, también existe otra relación de dependencia con `Fecha` a través de la fecha de creación de `CuentaBancaria`. Seguidamente se muestra el diagrama de clases correspondiente.



En el caso de la implementación de los métodos tenemos que el algoritmo para `Depositar` apenas se basa en sumar `valor` al `saldo`. Sin embargo, en `Extraer` se tiene que considerar el estado final donde el saldo nunca puede ser menor que el `saldoMínimo`. A continuación se muestra una implementación de esta clase.

```
public class CuentaBancaria
{
      public Persona propietario;
      public double saldo, saldoMínimo;
      public Fecha fechaCreación;

      public CuentaBancaria(Persona unaPersona, double unSaldo,
                            double saldoM)
      {
            propietario = unaPersona;
            saldo = unSaldo;
            saldoMínimo = saldoM;

            uint día = (uint)System.DateTime.Now.Day;
            uint mes = (uint)System.DateTime.Now.Month;
            uint año = (uint)System.DateTime.Now.Year;
            fechaCreación = new Fecha(día, mes, año);
      }

      public void Depositar(double valor)
```

```
        {
              saldo += valor;
        }

        public void Extraer(double valor)
        {
              if (saldo - valor >= saldoMínimo)
                  saldo -= valor;
        }
}
```

Como era de esperar al analizar el constructor de la clase `CuentaBancaria`, se aprecia que recibe tres parámetros con los cuales tiene que inicializar sus respectivos campos: `unSaldo` y `saldoM` que son asignados a `saldo` y `saldoMínimo` respectivamente, así como una referencia a un objeto `Persona` que es asignada a `propietario`.

En la sección anterior se vio que no siempre es sensato inicializar un campo de objeto a través de la asignación de una instancia creada previamente al llamado al constructor, pues se pueden crear situaciones de errores difíciles de determinar. Ahora bien, no siempre estas situaciones se pueden resolver tan fácilmente como en II.2.1, pues en este caso por ejemplo equivaldría a sustituir el parámetro `Persona unaPersona` del constructor de `CuentaBancaria`, por los necesarios para poder crear dentro del referido constructor la propia instancia de `Persona` para que solamente fuera referenciada por el campo `propietario`.

Sin dudas que de forma general existen situaciones que implicarían listas de parámetros inmensas, por ejemplo en este caso la lista crecería de 3 a 7 parámetros pues el cambio de `Persona unaPersona` por los valores necesarios para crear una instancia de `Persona` indica valores para `nombre`, `apellidos`, `día`, `mes` y `año` (los tres últimos para la fecha de nacimiento).

Pregunta: ¿Qué sucede si se tiene un constructor de una clase que necesita tres objetos de la clase `CuentaBancaria`?

Obviamente si se desea pasar los valores para crear cada una de las instancias la lista de parámetros seria inmensa. Todo esto indica que en el futuro se debe ser cuidadoso al definir los parámetros de los constructores.

Como interesante también en el código objeto de análisis es posible señalar que no es necesario pasar un parámetro para inicializar la variable pues la misma es creada a partir de la fecha del sistema utilizando las respectivas clases de la BCL (`System.DateTime`).

II.2.2 Variables y métodos de clase

Situación de análisis

En la definición de la clase `CuentaBancaria` se ha incluido una variable (`saldoMínimo`) para almacenar el valor mínimo que puede contener el saldo (`saldo`) que puede tener una instancia de esta clase. Esto quiere decir que para crear cada una de las instancias de `CuentaBancaria`, se debe pasar a través del constructor un valor (parámetro); no existiendo en principio forma de controlar que todas las instancias tengan el mismo valor en el referido campo como exige el dominio del problema.

En un primer intento de solución se puede estar pensando en asignarle un valor constante a la variable `saldoMínimo`, pero en ese caso todos los bancos tendrían que utilizar este valor sin posibilidades de modificación.

Analizando la problemática desde otra perspectiva: hasta el momento todas las variables y métodos que se han definido en las respectivas clases tienen sentido solamente para *instancias* u *objetos* de dichas clases, es por ello que este tipo de componente recibe el nombre de *variable de instancia* o *método de instancia* respectivamente. Pero este no es el "único tipo de variables o métodos" que existe, en el modelo OO aparecen otras componentes que no necesitan de instancias, es decir tienen que ser referenciadas a nivel de clase y por ello reciben el nombre de *variables* o *métodos de clase*. Particularmente las variables de clase comparten su valor con todas las instancias. Este es precisamente el caso de la variable `saldoMínimo` pues todos los

clientes de un banco deben tener precisamente el mismo valor para esta variable; es por ello que esta responsabilidad tiene que ser de la clase y no de las instancias.

Sintácticamente, los *componentes de clase* (variables y métodos por el momento) se expresan a través del modificador **`static`** como se muestra a continuación:

```
public class CuentaBancaria
{
      static public double saldoMínimo;

      public CuentaBancaria(Persona unaPersona, double unSaldo) ...

}
```

Pregunta: ¿Por qué en esta nueva definición de la clase `CuentaBancaria` no existe un parámetro para el saldo mínimo en el constructor?

Porque los parámetros que se definen en los constructores son principalmente con el objetivo de inicializar las variables de las instancias y ya se abordó con anterioridad que el objetivo de estos componentes de clase es que sean referenciados por las clases y no por las instancias; en el caso particular de las variables de clase solamente pueden ser modificadas por las clases. Por ejemplo, como se muestra a continuación para el caso de la clase `CuentaBancaria`:

```
CuentaBancaria.saldoMínimo = 100;
```

Vale destacar que todos los métodos de una clase tienen acceso directamente a los *componentes de clase* de su respectiva clase. Por ejemplo, como se muestra a continuación en el método `Extraer`:

```
public void Extraer(double valor)
{
      if (saldo - valor >= saldoMínimo)
            saldo -= valor;
}
```

Pregunta: ¿Cuál será la reacción del compilador de C# ante las siguientes instrucciones?

```
CuentaBancaria cuentaJuan =
 new CuentaBancaria(new Persona(...), 5000);

cuentaJuan.saldoMínimo = 100;
```

Inmediatamente reporta un mensaje similar al siguiente: `No se puede obtener acceso al miembro estático 'CuentaBancaria.saldoMínimo' con una referencia de instancia; utilice un nombre de tipo en su lugar`.

Para acceder a un componente o miembro de clase ya no es válida la sintaxis hasta ahora vista de `<objeto>.<responsabilidad>`, esto es lo que justifica el mensaje enviado por el compilador debido a la presencia de la instrucción `cuentaJuan.saldoMínimo = 100`. La sintaxis a usar para acceder a estos miembros será `<nombreClase>.<responsabilidad>`, como se muestra con anterioridad en el ejemplo donde se asigna el valor 100 a la variable `saldoMínimo` de la clase `CuentaBancaria` definida anteriormente (`CuentaBancaria.saldoMínimo = 100`).

Pregunta: ¿Cuál será la reacción del compilador de C# ante la definición de la siguiente clase?

```
public class A
{
    public int x;
    static public void Incrementa()
    {
          x++;
    }
}
```

Inmediatamente reporta un mensaje similar al siguiente: `Se requiere una referencia a objeto para el campo, método o propiedad no estáticos 'A.x'`

Es importante destacar que si se define un método de clase (`Incrementa`), entonces en el código del mismo sólo se podrá acceder implícitamente (sin sintaxis `<clase>.<miembro>`) a otras variables o métodos de clase del tipo de dato al que pertenezca. O sea, no se podrá acceder ni a las variables y métodos de instancia de la clase en que esté definido ni se podrá usar **`this`** ya que el método no se puede asociar a ninguna instancia.

El uso de las variables de clase no está tan generalizado ni es tan frecuente como el de los métodos de clase que inconscientemente se están utilizando desde que se realizaron las primeras implementaciones. Entre los métodos de clase que se han utilizado hasta el momento se encuentran:

```
Environment.TickCount();
Console.WriteLine();
Console.ReadLine();
```

También se han utilizado otros componentes de clase como `Math.PI` y `DateTime.Now.Year`. Del primero se tiene claridad que define una constante, por lo tanto se puede afirmar que esta es una *constante de clase*. El segundo parece una variable porque no termina con paréntesis como los métodos, pero a su vez según el estilo de código que se ha definido, no parece una variable porque las variables deben comenzar con minúscula, ni una constante porque no está completamente en mayúsculas; es decir, esta es una construcción del lenguaje que aún no ha sido abordada y que denota un nuevo *miembro de clase* que será abordado mas adelante.

II.2.2.1 Caso de estudio: Interfaz para la creación de objetos

Ejemplo: Escribir una aplicación que nos permita crear dos instancias de la clase `CuentaBancaria`.

```
class InstanciasCuentaBancaria
{
    static void Main(string[] args)
    {
          Console.WriteLine("Datos de la primera cuenta bancaria");
          Console.WriteLine("Datos del propietario");
          Console.Write("Nombre: ");
          string nombre = Console.ReadLine();
          Console.Write("Apellidos: ");
          string apellidos = Console.ReadLine();
          Console.WriteLine("Fecha de nacimiento");
          Console.Write("Día: ");
          uint d = uint.Parse(Console.ReadLine());
          Console.Write("Mes: ");
          uint m = uint.Parse(Console.ReadLine());
          Console.Write("Año: ");
          uint a = uint.Parse(Console.ReadLine());
          Console.Write("Saldo: ");
          uint saldo = uint.Parse(Console.ReadLine());

          Persona p = new Persona(nombre, apellidos, d, m, a);
          CuentaBancaria cuenta1 = new CuentaBancaria(p, saldo);
          Console.WriteLine();

          Console.WriteLine("Datos de la segunda cuenta bancaria");
          Console.WriteLine("Datos del propietario");
          Console.Write("Nombre: ");
          nombre = Console.ReadLine();
          Console.Write("Apellidos: ");
          apellidos = Console.ReadLine();
```

```
            Console.WriteLine("Fecha de nacimiento");
            Console.Write("Día: ");
            d = uint.Parse(Console.ReadLine());
            Console.Write("Mes: ");
            m = uint.Parse(Console.ReadLine());
            Console.Write("Año: ");
            a = uint.Parse(Console.ReadLine());
            Console.Write("Saldo: ");
            saldo = uint.Parse(Console.ReadLine());

            p = new Persona(nombre, apellidos, d, m, a);
            CuentaBancaria cuenta2 = new CuentaBancaria(p, saldo);
            Console.WriteLine(cuenta2.ToString());
      }
   }
```

Nota: En el caso particular de esta solución se ha presentado la aplicación completa porque se tienen intereses didácticos con la clase `InstanciasCuentaBancaria` y su método `Main`.

Es evidente que la complejidad y dimensiones de la solución anterior se deben determinantemente a la lectura o entrada de los datos para la creación de los objetos y de hecho se aprecia que por supuesto los algoritmos de creación de ambas cuentas son muy similares. A continuación se presenta una variante introduciendo un método que permite reutilizar código para simplificar la solución.

```
class InstanciasCuentaBancaria
{
      static CuentaBancaria LeerCuentaBancaria()
      {
            Console.WriteLine("Datos del propietario");
            Console.Write("Nombre: ");
            string nombre = Console.ReadLine();
            Console.Write("Apellidos: ");
            string apellidos = Console.ReadLine();
            Console.WriteLine("Fecha de nacimiento");
            Console.Write("Día: ");
            uint d = uint.Parse(Console.ReadLine());
            Console.Write("Mes: ");
            uint m = uint.Parse(Console.ReadLine());
            Console.Write("Año: ");
            uint a = uint.Parse(Console.ReadLine());
            Console.Write("Saldo: ");
            uint saldo = uint.Parse(Console.ReadLine());

            Persona p = new Persona(nombre, apellidos, d, m, a);
            return new CuentaBancaria(p, saldo);
      }

      static void Main(string[] args)
      {
            Console.WriteLine("Datos de la primera cuenta bancaria");
            CuentaBancaria cuenta1 = LeerCuentaBancaria();

            Console.WriteLine();

            Console.WriteLine("Datos de la segunda cuenta bancaria");
            CuentaBancaria cuenta2 = LeerCuentaBancaria();
      }
}
```

A través de la inclusión de este nuevo método se ha separado la interfaz y creación de cuenta en un módulo por separado, que sólo se encarga de leer los valores necesarios para crear un objeto de tipo `CuentaBancaria` que luego es devuelto a través de la instrucción **`new`** `CuentaBancaria(…)` en la cual primero se crea el objeto a través del operador **`new`** y luego la referencia devuelta por este es a su vez retornada por **`return`**. Solamente faltaría por utilizar el método descrito, tantas veces como instancias se desee crear.

Note en este caso como el método `Main` se ha simplificado considerablemente con la definición del método `LeerCuentaBancaria`.

Pregunta: ¿Es obligatoria la definición del método `LeerCuentaBancaria` como miembro de clase (**`static`**)?

La definición de este método como miembro de clase es obligatoria porque el mismo es llamado dentro de un método precisamente de clase (**`static void`** `Main(...)`)

Sin dudas que este refinamiento ha mejorado la solución considerablemente pero es muy particular para esta aplicación. Es decir, si se está desarrollando otra aplicación y es necesario nuevamente crear algunas instancias de la clase `CuentaBancaria` cuyos datos serían entrados por teclado, entonces se debe definir un método semejante a `LeerCuentaBancaria`.

Una variante mucho más general se tiene si se define un método de clase que se denominará `Leer` en la propia clase `CuentaBancaria` (semejante al anteriormente descrito), entonces de esta forma al menos se dispondrá de un recurso siempre que se necesite crear instancias de la referida clase, como se muestra seguidamente:

```
public class CuentaBancaria
{
      public CuentaBancaria(Persona unaPersona, double unSaldo)
      public static CuentaBancaria Leer()
      {
            Console.WriteLine("Datos del propietario");
            Console.Write("Nombre: ");
            string nombre = Console.ReadLine();
            Console.Write("Apellidos: ");
            string apellidos = Console.ReadLine();
            Console.WriteLine("Fecha de nacimiento");
            Console.Write("Día: ");
            uint d = uint.Parse(Console.ReadLine());
            Console.Write("Mes: ");
            uint m = uint.Parse(Console.ReadLine());
            Console.Write("Año: ");
            uint a = uint.Parse(Console.ReadLine());
            Console.Write("Saldo: ");
            uint saldo = uint.Parse(Console.ReadLine());
            Persona p = new Persona(nombre, apellidos, d, m, a);
            return new CuentaBancaria(p, saldo);
      }
}
```

Con este refinamiento de la clase `CuentaBancaria` se puede entonces, a través de la siguiente secuencia de instrucciones crear dos instancias de la referida clase.

```
{
      Console.WriteLine("Datos de la primera cuenta bancaria");
      CuentaBancaria cuenta1 = CuentaBancaria.Leer();
      Console.WriteLine("Datos de la segunda cuenta bancaria");
      CuentaBancaria cuenta2 = CuentaBancaria.Leer();
}
```

Claro está que con esta solución no están resueltos todos los problemas porque si un usuario de la clase `CuentaBancaria` no gusta de la interfaz que se propone entonces el mismo tendrá que diseñar su propia interfaz de lectura para crear las instancias.

II.2.3 Ejercicios

1. Determine los mensajes que se muestran al ejecutar el siguiente bloque de código. Ilustre gráficamente el proceso de creación de los objetos presentes.

```
{
   Fecha f = new Fecha(1, 1, 1980);
   Persona juan = new Persona("Juan", "Pérez", f);
   Persona ana = new Persona("Ana", "García", f);
   Console.Writeln(juan.ToString());
   Console.Writeln(ana.ToString());

   f = new Fecha(1, 1, 1980);
   juan = new Persona("Juan", ana.apellidos, f);
   ana = new Persona("Ana", juan.apellidos, f);
   f = new Fecha(2, 2, 1981);
   Console.Writeln(juan.ToString());
   Console.Writeln(ana.ToString());
}
```

2. Ilustre gráficamente el proceso de creación de las siguientes instancias de `Cuenta Bancaria`:

```
{
   Fecha f = new Fecha(1, 1, 1980);
   Persona prop = new Persona("Juan", "Pérez", f);
   CuentaBancaria cuentaJuan = new CuentaBancaria(prop, 100, f);

   f = new Fecha(2, 2, 1985);
   prop = new Persona("Ana", "Fdez”, f);
   CuentaBancaria cuentaAna = new CuentaBancaria(prop, 100, f);
}
```

3. Muestre los mensajes que imprime el siguiente segmento de código. Ilustre gráficamente el resto de las operaciones:

```
{
   Fecha f = new Fecha(1, 1, 1980);
   Persona prop = new Persona("Juan", "Pérez", f);
   CuentaBancaria cuentaJuan = new CuentaBancaria(prop, 100);
   f = new Fecha(2, 2, 1985);
   prop.Nombre = "Pedro";
   CuentaBancaria cuentaPedro = new CuentaBancaria(prop, 200);
   Console.WriteLine(cuentaJuan.ToString());
   Console.WriteLine(cuentaPedro.ToString());

   cuentaJuan.Propietario.Nombre = "Pedro";
   cuentaPedro.Propietario.Nombre = "Juan";
   f = new Fecha(3, 3, 1987);
   Console.WriteLine(cuentaJuan.ToString());
   Console.WriteLine(cuentaPedro.ToString());
}
```

4. Diseñe e implemente una clase que permita representar automóviles a través de las siguientes responsabilidades: marca, modelo, fecha de fabricación y propietario (`Persona`).

5. En el método `Leer` de la clase `CuentaBancaria` definido en la sección II.2.2.1 se realiza de forma explícita la lectura de los datos para crear las instancias de `Persona` y `Fecha`. Defina un método `Leer` en las clases `Persona` y `Fecha`, posteriormente refine el método `Leer` de la clase `CuentaBancaria`.

6. Diseñe e implemente una clase para representar segmentos de recta (`SegmentoRecta`) (auxíliese de la clase `PuntoR2` propuesta en I.1), permita obtener la longitud de los mismos. También implemente métodos para determinar la distancia de un `PuntoR2` a un `SegmentoRecta` y para determinar si un `PuntoR2` está contenido en un `SegmentoRecta` (a través de diferentes algoritmos).

7. Ilustre gráficamente el proceso de creación de los siguientes objetos:

```
{
   Punto P1 = new Punto(0, 0);
   Punto P2 = new Punto(0, 5);
   SegmentoRecta s1 = new SegmentoRecta(P1, P2);
   P2 = new Punto(5, 0);
   SegmentoRecta s2 = new SegmentoRecta(P1, P2);
}
```

8. Diseñe e implemente otras figuras geométricas en el plano y en el espacio.

9. Defina métodos `Leer` (similares al que fue implementado en la clase `CuentaBancaria`) en las diferentes clases implementadas los capítulos precedentes.

10. Modifique la clase Persona implementada en este capitulo de forma tal que permita representar los miembros de una familia (`apellidos` común).

11. Proponga una estrategia para determinar cuantas instancias de una clase se tienen en tiempo de ejecución.

12. Diséñe e implemente una clase que permita representar a un profesor de la Universidad, del que se conoce nombre, apellido, categoría docente, años de experiencia, fecha de nacimiento y asignatura que imparte. De dicha asignatura se conoce el nombre, cantidad de horas clases y si tiene examen final. Garantice entre otras las responsabilidades que le permitan a un profesor cambiar la asignatura que imparte a sabiendas que este proceso solo puede efectuarse si la cantidad de horas de la asignatura que esta siendo impartida por el profesor es menor que la que se le desea asignársele y si la misma tiene examen final.

13. Redefina la clase `Triángulo` esta vez representándolo a través de las coordenadas de sus vértices. Teniendo en cuenta este cambio, redefina los métodos según sea necesario. Exprese las coordenadas de los vértices utilizando la clase `PuntoR2`.

II.2.4 Bibliografía complementaria

- Capítulos 5, 7 y 8 de Jose A. González Seco, El lenguaje de programación C#, 2002. Puede descargarse en http://www.josanguapo.com/

Capítulo II.3

II.3 Encapsulamiento. Programación por contratos. Excepciones

El diseño de la abstracción de las clases de la manera más útil para que los programadores las usen es de suprema importancia al desarrollar software reutilizable [13]. A partir de lo anterior es posible deducir que si se desarrolla una interfaz estable y estática que permanezca inalterable a través de los cambios de la implementacion, cada vez menos se necesitará modificar las aplicaciones a lo largo del tiempo.

El mecanismo de encapsulamiento (encapsulación) o principio de la ocultación de la información, le permite a los diseñadores de clases determinar qué miembros de estas pueden ser utilizados por otros programadores y cuáles no. A su vez este mecanismo nos permite ocultar todos los detalles relativos a la implementación interna y sólo dejar visibles aquellos que se puedan usar con seguridad y le facilita al creador la posterior modificación. Este mecanismo le permite a las clases proporcionar una interfaz con las responsabilidades que los clientes pueden acceder directamente.

Una parte importante en el diseño de una interfaz de clase es tener un conocimiento lo suficientemente profundo del dominio del problema. Este conocimiento le ayudará a crear una interfaz que de al usuario acceso a la información y métodos, a la vez que le aísla de los procedimientos internos de la clase. Necesita diseñar una interfaz no solamente para resolver problemas de hoy, sino para también abstraer lo suficiente los aspectos internos de la clase de forma que los miembros privados de ésta puedan someterse a cambios ilimitados sin que afecte el código existente [13].

Otro elemento muy importante del diseño de la abstracción de la clase es tener en mente en todo momento a los futuros clientes. Esto sin dudas nos permite determinar qué miembros de la clase deben ser accesibles públicamente [13].

Otro elemento a destacar en la necesidad e importancia del principio de encapsulamiento es el control de la integridad de los objetos, es decir que se puedan crear objetos del dominio en cuestos y una vez creados los objetos, éstos permanezcan siempre perteneciendo a dicho dominio.

La programación por contratos ve las relaciones entre las clases y sus clientes como un acuerdo formal, que expresa los derechos y obligaciones de cada parte. Aún cuando C# por el momento no posee mecanismos formales para expresar la programación por contratos se utilizán las excepciones como mecanismo homogéneo para el caso en que los clientes de las clases no cumplan con el acuerdo formal establecido.

II.3.1 Integridad de los objetos

Situación de análisis

Observe detenidamente el bloque de código que se presenta seguidamente y reflexione al respecto.

```
{
      Fecha fecha1 = new Fecha(1, 1, 1975);

      fecha1.día = 32;

      fecha1.día = 30;
      fecha1.mes = 2;

      Fecha fecha2 = new Fecha(31, 4, 1969);
}
```

Luego de analizar el código, se observa que la variable `fecha1` ha transitado por cuatro estados y de ellos en dos oportunidades contiene objetos que no pertenecen al dominio de los objetos `Fecha`, o lo que es equivalente a decir, la integridad del objeto `fecha1` se ha corrompido en dos oportunidades. En el caso de la variable `fecha2` desde su creación no representa a un objeto del dominio `Fecha`.

Pregunta: ¿Qué elementos han facilitado que se corrompa la integridad de los objetos `Fecha`?

> En este caso se tienen dos respuestas, esto se produce porque existe total acceso a las variables de los objetos `Fecha` y porque el constructor "no se preocupa" por verificar que se garantice la integridad durante la creación del objeto, es decir, los objetos se crean independientemente de que los valores de inicialización no se correspondan con datos que garanticen objetos del dominio `Fecha`.

En un primer refinamiento se limita el acceso a las variables de las instancias de la clase `Fecha`. Esta limitación se traduce en no permitir acceder directamente a las variables de dichos objetos (utilizando la sintaxis `<objeto>.<responsabilidad>`). Este primer refinamiento no permitirá acciones como: `fecha1.día = 32`.

Este control sobre el acceso a los campos se logra a través del mecanismo de encapsulamiento mencionado al inicio del presente capítulo y en capítulos precedentes y que constituye uno de los conceptos básicos del paradigma OO.

El encapsulamiento se consigue añadiendo modificadores de acceso en las definiciones de los miembros y tipos de datos. Estos modificadores constituyen palabras reservadas del lenguaje que se les anteponen para indicar desde que código puede accederse a ellos, entendiendo por acceder cualquier cosa que no sea definirlo.

En caso de no especificarse modificador alguno, se considera por defecto u omisión que los miembros de un tipo de dato sólo son accesibles desde código situado dentro de la definición del mismo tipo. Aunque también puede especificarse y para ello ha de utilizarse uno de estos modificadores de C# que armonizan con las especificaciones estándares del modelo OO:

- **public**: indica que la componente puede ser accedida desde cualquier código (+).
- **private**: sólo puede ser accedido desde el código de la clase a la que pertenece. Es lo considerado por defecto (-).
- **protected**: permite el acceso desde el código de la clase a la que pertenece o de subclases suyas (#).

Existe una cuarta especificación de visibilidad (**`internal`**) muy particular de C# que garantiza la visibilidad de las componentes a nivel de ensamblados.

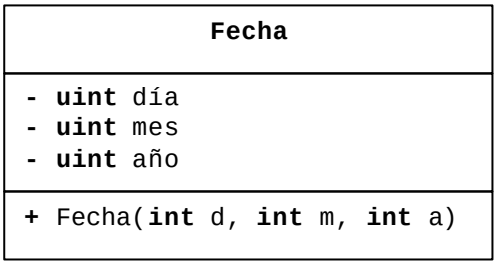
A continuación se presenta una aplicación de estos niveles de visibilidad precisamente en la clase `Fecha`.

```
public class Fecha
{
      private int día, mes, año;

      public Fecha(int d, int m, int a)...

}
```

En C#, el nivel de visibilidad por defecto es **`private`**. Es por ello que el código que se muestra a continuación es equivalente al que se acaba de presentar.

```
public class Fecha
{
      int día, mes, año;

      public Fecha(int d, int m, int a)...

}
```

En el diagrama de clases estos niveles de visibilidad se representan de la siguiente forma.



A partir de este momento, las variables `día`, `mes` y `año` no están disponibles para los clientes de la clase `Fecha`, por lo tanto no es posible modificar sus valores, es decir, al intentar realizar la siguiente operación:

```
{
      Fecha fecha1 = new Fecha(1, 1, 1975);
      fecha1.día = 32;
}
```

Inmediatamente el compilador reportaría un error `similar al siguiente: 'Fecha.día' no es accesible debido a su nivel de protección`

Es importante notar en el refinamiento de la clase `Fecha` mostrado con anterioridad, que al constructor de la clase se le especifica un nivel de visibilidad público. Por lo general no tiene mucho sentido declarar una clase con todos sus constructores privados, puesto que de esta forma no es permisible crear instancias de dicha clase desde código externo a la clase, salvo que ese sea el propósito deseado, aunque más adelante se verá que existen otras formas de especificar o lograr ese propósito.

El hecho de que en C# el nivel de visibilidad por defecto es **`private`**, ha sido la causa de que a las responsabilidades de las clases definidas en los capítulos precedentes hayamos tenido que colocarle **`public`** delante de las componentes miembros para poder tener acceso a las mismas posteriormente.

Pregunta ¿Cómo podría ahora el `Estudiante juan` suministrarle la información a un colega del grupo sobre su fecha de nacimiento?

Esto no es posible, pues se le oculta por completo a los clientes el acceso a las responsabilidades de `Fecha` que almacenan sus respectivos valores.

Una variante de solución a esta problemática sería a través de la definición de métodos de acceso que retornan el valor de las respectivas variables. Este es el mecanismo de acceso utilizado en la mayoría de los lenguajes OO pero en la siguiente sección (II.3.2) se verá que C# ofrece además, un mecanismo aún más interesante. Seguidamente se muestra un refinamiento a la clase `Fecha` incorporándole los respectivos métodos de acceso.

```
public class Fecha
{
      int día, mes, año;
      public int GetDía()
      {
            return día;
      }
      public int GetMes()
      {
            return mes;
      }
      public int GetAño()
      {
            return año;
      }
}
```

Entonces a partir de este momento, se tendría acceso a los valores de las variables `día`, `mes` y `año` a través de sus respectivos métodos de acceso como se muestra seguidamente:

```
{
      fecha1.GetDía();
      fecha1.GetMes();
      fecha1.GetAño();
}
```

II.3.2 Propiedades

Situación de análisis

Supóngase que las formas que se presentan a continuación garantizan el acceso al contenido de las variables de las instancias de `Fecha`:

a) `fecha1.GetDía()`
b) `fecha1.Día()`
c) `fecha1.Día`

Pregunta ¿Cuál de las variantes que se muestra con anterioridad le impone mayor legibilidad al acceso?

Sin dudas que a través de la opción c), note que este acceso es tan natural como el acceso a cualquier variable, se supone que no es una variable porque comienza con mayúscula y no coincide con el estilo de código propuesto para las variables.

Los recursos similares al utilizado en el inciso c) se denominan *propiedades*.

Según Archer [13], utilizar métodos de acceso funciona bien y es una técnica utilizada por los programadores de varios lenguajes OO, incluidos C++ y Java. Sin embargo, C# proporciona un mecanismo más rico incluso, las *propiedades*, que tienen las mismas capacidades como métodos de acceso y que son mucho más elegantes para el cliente. Mediante las propiedades, un programador puede escribir un cliente que pueda acceder a las variables de una clase como si fueran públicas sin saber si existe un método de acceso.

Para definir una propiedad se usa la siguiente sintaxis:

```
[<modificadores>] <tipoPropiedad> <nombrePropiedad>
{
      get
      {
            <códigoEscritura>
      }
      set
      {
            <códigoEscritura>
      }
}
```

Una propiedad definida de esta forma va a ser 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>`.

La forma de acceder a una propiedad, indistintamente para lectura o escritura, es la misma que se usa para acceder a un campo de su mismo tipo.

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.

Como se ha dicho con anterioridad, cuando se acceda para escribir, se ejecuta el bloque de código prefijado en el bloque **`set`** de la propiedad en cuestión, dentro de este código **`set`** se puede hacer referencia

al valor que se solicita asignar a través de un parámetro especial del mismo tipo de la propiedad llamado **value**, por lo que se considera erróneo definir una variable con ese mismo nombre, pues se perdería el acceso a este parámetro especial brindado por **set**.

De forma análoga cuando se desee acceder a la propiedad para leerla, se ejecutara el bloque **get**, que sera el encargado de devolver el valor deseado siendo necesario que dentro de este código se utilice la cláusula **return** para devolver un objeto del tipo de dato de la propiedad.

Seguidamente se muestran ejemplos de propiedades para acceder a cada una de las variables de la clase `Fecha`.

```
public class Fecha
{
      int día, mes, año;

      public uint Día
      {
            get
            {
                  return día;
            }
      }
      public uint Mes
      {
            get
            {
                  return mes;
            }
      }
      public uint Año
      {
            get
            {
                  return año;
            }
      }
}
```

Note que las propiedades descritas con anterioridad no cuentan con especificaciones del tipo **set**, por lo que constituirán propiedades de solo lectura, o lo que es equivalente, sólo se podrá acceder a ellas para leer su valor. Aunque en el ejemplo se ha optado por asociar un campo privado a cada propiedad, en realidad nada obliga a que ello se haga y es posible definir propiedades que no tengan campos privados.

Ejemplo: Obtener una aplicación de consola que permita leer una instancia de `Fecha` e imprima en que semestre se encuentra.

Como se expuso con anterioridad, la forma de acceder a una propiedad es exactamente la misma que se usaría para acceder a un campo de su mismo tipo. Luego, para determinar en que semestre se encuentra una instancia de `Fecha` apenas se tiene que leer su propiedad `Mes` y luego comparar con 6 como se muestra seguidamente:

```
{
    Fecha f = Fecha.Leer();

    if (f.Mes < 6)
          Console.WriteLine(“Primer semestre“) ;
    else
          Console.WriteLine(“Segundo semestre“) ;
}
```

Nota: Se ha utilizado el método `Leer` de la clase `Fecha` para simplificar el código de aplicaciones, esto será un hábito a partir de este momento con el objetivo de centrar la atención en los elementos novedosos y más significativos.

Las propiedades constituyen una mezcla entre el concepto de campo de dato y el concepto de método. Éstas no almacenan datos pero se utilizan también como recurso de acceso a campos de datos lo que hace que se utilizan como si los almacenase. 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. Un importante uso de este código es para garantizar la integridad de los objetos.

II.3.3 Integridad inicial de los objetos

Situación de análisis

Hasta el momento, a través del concepto de encapsulamiento, se ha garantizado que una vez creadas correctamente las instancias de la clase `Fecha` no pierdan su integridad porque no es posible modificar sus valores.

Nótese el destaque de la premisa: *una vez creadas correctamente las instancias de la clase*. Recordando el análisis desarrollado alrededor de la situación de análisis de la sección II.3.1.1 y teniendo en cuenta que hasta el momento no se ha realizado ninguna modificación al constructor de dicha clase entonces será permitido la creación de la instancia `fecha2` que como se había visto con anterioridad no constituye un objeto del dominio `Fecha`.

Pregunta: ¿Por qué es posible que se haya creado una instancia de `Fecha` (`fecha2`), que sin embargo, no constituye un objeto de este dominio o clase?

Dos elementos han tenido que ver en esta situación, en primer lugar el cliente no ha cumplido con su responsabilidad de pasarle un juego de valores correctos (31/4/1975) al constructor de la clase `Fecha` y en segundo lugar el constructor de `Fecha` no se ha preocupado por verificar o comprobar esta situación y ha creado el objeto de cualquier manera.

II.3.3.1 Programación por contratos

A partir del análisis de la sección anterior es posible concluir que evidentemente se necesita de un mecanismo para establecer y/o verificar un compromiso o acuerdo formal entre las clases y sus clientes. Es aquí donde juega su papel determinante la *programación por contratos*.

El concepto de programación por contratos no es más que el compromiso mutuo que surge entre las clases y los clientes de las mismas que conllevará al éxito de la aplicación si cada parte cumple con su compromiso en este proceso. Los implementadores de las clases por su parte, tienen que lograr que la misma cumpla con su prometido si sus usuarios cumplen disciplinadamente con los requisitos de utilización de la misma. Y por otra parte, los clientes de la clase tienen que usarla en la forma que se le pide esperando de estas que no le generen errores si cumplen con los requisitos.

Corresponde ahora refinar el constructor de la clase `Fecha` para que restrinja la creación de objetos solamente a ejemplares del dominio Fecha. Note que en este caso es una responsabilidad de los clientes de la clase Fecha garantizar que los valores suministrados al constructor se correspondan efectivamente con ternas que conformen objetos del referido dominio.

Al referirnos a responsabilidades que tiene que cumplir el cliente se comienzan a establecer relaciones con la programación por contratos que se abordará muy superficialmente a continuación teniendo en cuenta la no existencia de mecanismos formales en C# por el momento para expresar este concepto.

La correctitud de un método es algo relativo, y depende en gran parte de la especificación. Una fórmula de corrección (o tripleta de Hoare) es una expresión de la forma:

(1) `{P} A {Q}`

La fórmula de corrección anterior se lee: una ejecución de `A` que comience en un estado en el que se cumpla `P` terminará en un estado en el que se cumple `Q`. Aquí `A` denota un método. `P` y `Q` se denominan aserciones, de las cuales `P` será la pre-condición y `Q` será la post-condición.

La pre-condición establece las propiedades que se tienen que cumplir cada vez que se llame al método. La post-condición establece las propiedades que debe garantizar la rutina cuando retorne. Ejemplo:

```
{x >= 9}   x = x + 5   {x >= 13}
```

En este caso, una post-condición más fuerte sería `{x >= 14}`. En este mismo caso, una pre-condición más débil sería `{x >= 8}`. El hecho de que una condición se considere más fuerte que otra implica que esta última restringe menos el conjunto solución que la primera.

La propiedad de que un método satisfaga su especificación, si termina, se conoce como corrección parcial. Si un programa satisface su especificación y termina, se dice que es totalmente correcto (total corrección implica además terminación).

Las aserciones permiten, a quienes desarrollan software, construir programas correctos, y documentar porqué son correctos.

En el ejemplo del constructor de la clase `Fecha` la pre-condición es que la terna `d`, `m` y `a` se corresponda correctamente con una fecha y la post-condición es que se cree correctamente un objeto `Fecha` con los valores `día = d`, `mes = m` y `año = a`.

Definir una pre-condición y una post-condición para un método es una forma de definir un contrato que liga a dicho método con quienes le invocan.

En (1), el método `A` le está diciendo a sus clientes: "si usted me promete que al llamarme se satisface `P`, entonces yo le prometo entregar un estado final en el que se satisface `Q`".

Para los contratos entre clases la pre-condición compromete al cliente: define las condiciones bajo las cuales es legítima la llamada al método. Es una obligación para el cliente y un beneficio para el proveedor mientras que la post-condición compromete a la clase (el proveedor): define las condiciones que debe asegurar la rutina al retornar. Esto es un beneficio para el cliente y una obligación para el proveedor.

El uso de contratos simplifica la programación, en los lenguajes que implementan la programación por contratos el programador puede dar por supuesto que las pre-condiciones se cumplen, sin tener que verificarlas.

En los lenguajes que no implementan este concepto se tiene que llevar a cabo una programación más defensiva, es decir las pre-condiciones se deben verificar dentro de las propios métodos y garantizar a conciencia el cumplimiento de las post-condiciones. En el listado siguiente se muestra otro refinamiento de la clase `Fecha` a partir de la verificación de las pre-condiciones en su constructor.

```
public class Fecha
{
      uint día, mes, año;
      public Fecha(uint d, uint m, uint a)
      {
            bool fechaOK = false;
            if (m > 0 && m < 13)
            {
                  mes = m;
                  año = a;
                  if (d > 0 && d <= DíasMes())
                  {
                        día = d;
                        fechaOK = true;
                  }
            }
            if (!fechaOK)
            {
                  día = 1;
                  mes = 1;
                  año = 1980;
```

```
            }
        }
}
```

En este caso la precondición que tienen que cumplir los clientes de la clase `Fecha` (y que será verificada por el constructor) es que la terna de valores `día/mes/año` a pasar en el constructor como parámetro, permita formar una instancia de dicho dominio de objetos. La post-condición a cumplir por el constructor de `Fecha` es garantizar la creación correcta del objeto.

Nótese en este refinamiento que si el cliente cumple con la pre-condición entonces el constructor garantiza que se cree el objeto correctamente, es decir cumple con la post-condición. Pero si el cliente no cumple con su parte del contrato entonces el constructor tampoco puede garantizar la suya. En este caso, cuando el cliente no cumple la pre-condición, el constructor ha decidido crear un objeto con valores preestablecidos (1/1/1980) que garanticen la integridad del mismo.

Pregunta: ¿Es una solución rigurosa la adoptada por el constructor en caso de no cumplirse la pre-condición?

Note que el constructor ha adoptado una actitud muy conservadora pues aún cuando el cliente no ha cumplido con su parte del contrato, se crea un objeto y no se notifica al mismo de su incumplimiento del acuerdo.

Existen varias tendencias o filosofías para el tratamiento o detección de errores, una es el llamado *retorno silencioso*, denominado de esta forma pues ante la incapacidad de llevar a cabo una servicio solicitado se opta por no hacer nada y retornar, lo que es considerado una bomba de tiempo que puede explotar tarde o temprano; otra filosofía es la *histeria*, la cual consiste en detectar el error, enviar algún mensaje que indique lo que ha ocurrido y terminar la ejecución del programa radicalmente. La práctica más honesta es combinar estas dos filosofías, es decir, dar la alarma pero organizadamente [3].

Una solución más homogénea y rigurosa en el caso de los constructores podría ser que si los clientes no cumplen con la parte que les corresponde entonces los objetos no son creados y se le da una notificación a quien invocó dicho constructor.

II.3.3.2 Constructores privados

Situación de análisis

De momento una vez que se está ejecutando un constructor no se conoce como evitar que el mismo cree una instancia.

Pregunta: ¿Existe solución a este problema con los recursos presentados hasta el momento?

Una solución a esta problemática podría ser definiendo los constructores como privados e implementando un método de clase (público por supuesto) a través del cual se invocarían los constructores, solamente en el caso que los parámetros de inicialización fueran correctos, en caso contrario se retorna **`null`** automáticamente.

Ejemplo: Refinar la clase `Fecha`, diseñar el algoritmo de implementación del método de clase referido con anterioridad.

La clase `Fecha` entonces incorpora el referido método de clase que se denominara `CrearInstancia` cuyo algoritmo solamente tiene que discriminar las ternas que no constituyen fechas (retornar **`null`**) de las correctas (invocar al constructor) como se muestra informalmente a continuación:

```
public class Fecha
{
    uint día, mes, año;
    Fecha(uint d, uint m, uint a) {...}
    public static Fecha CrearInstancia(uint d, uint m, uint a)
    {
          if <subdominio para discriminar fechas>
                return null;
          else
```

```
                return new Fecha(d, m, a);
        }
    }
```

Situación de análisis

Analice la siguiente expresión

```
(m == 0 || m > 13 || d == 0 || d > DíasMes())
```

como variante de implementación del subdominio

```
<subdominio para discriminar fechas>
```

Pregunta: ¿Es correcta la expresión anterior?

La expresión anterior forma parte de un método de clase y sin embargo, se intenta acceder a un miembro de instancia en el caso del método `DíasMes`, lo cual no es permitido porque como se vio en el capítulo anterior desde miembros de clase no se puede tener acceso a miembros de instancia.

Una variante de solución podría ser la creación de un método de clase `DíasMes` con un parámetro para el respectivo `mes` y otro método `DíasMes`, ahora de instancia que se auxilia de este. Sin embargo, en la implementación del método de clase `DíasMes` se accede al método `Bisiesto` que también es de instancia, luego para este último método habría que realizar un análisis similar, lo que implica que también se necesita un parámetro para el año en el método `DíasMes`. A continuación se presenta otro refinamiento de la clase `Fecha` incluyendo estos elementos:

```
public class Fecha
{
      uint día, mes, año;

      Fecha(uint d, uint m, uint a) {...}

      public static Fecha CrearInstancia(uint d, uint m, uint a)
      {
            if (m == 0 || m > 13 || d == 0 || d > DíasMes(m, a))
                  return null;
            else
                  return new Fecha(d, m, a);
      }

      static public uint DíasMes(uint mes, uint año) {...  }

      public uint DíasMes()
      {
            return DíasMes(mes, año);
      }

      public static bool bisiesto(uint año) {...}

      public bool bisiesto()
      {
            return bisiesto(año);
      }
}
```

Situación de análisis

La existencia del método de clase `CrearInstancia` facilita la definición de otro método de clase `Leer` para crear instancias de `Fecha` cuya implementación se muestra a continuación:

```
static public Fecha Leer()
{
      Console.Write("Día: ");
      uint d = uint.Parse(Console.ReadLine());
      Console.Write("Mes: ");
      uint m = uint.Parse(Console.ReadLine());
      Console.Write("Año: ");
      uint a = uint.Parse(Console.ReadLine());
      return CrearInstancia(d, m, a);
}
```

Pregunta: ¿Qué sucede al ejecutar las siguientes instrucciones?

```
{
     Fecha f = Fecha.Leer();

     Console.WriteLine(f.ToString());
}
```

para los siguientes juegos de datos.

a) `1/1/1980`

b) `30/2/2003`

c) `3.1/12/2003`

En el primer caso apenas se imprime el mensaje `1/1/1980` que se corresponde con la `Fecha` correctamente tecleada.

En los otros dos casos se detiene la ejecución del programa y se muestra un mensaje que comienza de la siguiente forma: `Excepción no controlada: System.NullReferenceException: Referencia a objeto no establecida como instancia de un objeto.`

Estos errores se producen porque el método `Leer` se auxilia de `CrearInstancia` para crear los objetos `Fecha` y en los dos últimos casos no se crean objetos en `f`, sino que esta variable almacena una referencia a **`null`** puesto que los datos no pertenecían al dominio de `Fecha`. Como consecuencia de esto cualquier intento de acceder a alguna de las responsabilidades de `f` se considerará un error (`f.ToString()`).

Pregunta: ¿Qué tipo de filosofía para el tratamiento de errores se ha puesto de manifiesto?

Retorno silencioso por parte del programador ya que apenas se retorna **`null`** en el caso de existir algún problema en los datos de creación de las instancias de `Fecha` e histeria por C# pues al detectar una acceso a un miembro de una variable **`null`** aborta la ejecución del programa. Es de señalar el peligro que se corre al utilizar estas filosofías, pues en cualquier momento existe la posibilidad de que la aplicación aborte por un error.

II.3.3.3 Excepciones

En C# no existen recursos del lenguaje para establecer la programación por contratos. En C# como en C++ y otros lenguajes se aplica una filosofía más sencilla y conocida como *dispara* y *captura* (*throw* & *catch*).

Las excepciones son el mecanismo recomendado en la plataforma .NET para la propagación de errores que se produzcan durante la ejecución de las aplicaciones (divisiones por 0, intentos de lectura de archivos dañados, etc.). Básicamente una excepción es un objeto (“descendiente” de la clase `System.Exception`)

que se genera cuando en tiempo de ejecución se produce algún error y que contiene información sobre el mismo.

"Las excepciones son condiciones de error que surgen cuando el flujo normal que sigue el código es poco práctico o imprudente" [13].

Una llamada a un método tiene éxito si termina su ejecución en un estado en el que satisface el contrato del mismo, fracasa o falla si no tiene éxito. Una excepción es un suceso en tiempo de ejecución que puede causar que un método fracase.

Como se había comentado, una excepción es un objeto y el mismo cuenta con campos que le permitan describir las causas del error y a cuyo tipo (el de la excepción) suele asociársele un nombre que resuma claramente su causa. Por ejemplo, para informar errores de división por cero se suele utilizar una excepción predefinida de tipo `DivideByZeroException`, en cuya propiedad `Message` se detallan las causas del error producido.

Para informar de un error no es suficiente crear el objeto del tipo de excepción apropiado, sino que hay que pasárselo al mecanismo de propagación de errores, esta acción se denomina lanzar la excepción y para hacerlo se utiliza la sintaxis:

```
throw <objetoExcepcionALanzar>;
```

Por ejemplo, para lanzar una excepción de tipo `DivideByZeroException` se podría hacer de la siguiente forma:

```
Exception e = new DivideByZeroException();
throw e;
```

ó

```
throw new DivideByZeroException();
```

Sin embargo, teniendo en cuenta los objetivos de este texto apenas serán lanzadas las excepciones de la siguiente forma:

```
throw new Exception(<mensajeError>);
```

A continuación se muestra una aplicación del concepto de excepción en un próximo refinamiento al constructor de la clase `Fecha` en el siguiente en el listado (ha sido incorporado también un método `Leer`).

```
public class Fecha
{
      private uint día, mes, año;

      public Fecha(uint d, uint m, uint a)
      {
            if (m > 0 && m < 13)
            {
                  mes = m;
                  if (d > 0 && d <= DíasMes())
                  {
                        día = d; año = a;
                  }
                  else
                        throw new Exception("Día incorrecto");
            }
            else
                  throw new Exception("Mes incorrecto");
      }

      static public Fecha Leer()
      {
```

```
            Console.Write("Día: ");
            uint d = uint.Parse(Console.ReadLine());
            Console.Write("Mes: ");
            uint m = uint.Parse(Console.ReadLine());
            Console.Write("Año: ");
            uint a = uint.Parse(Console.ReadLine());

            return new Fecha(d, m, a);
        }
}
```

En el ejemplo anterior se aprecia como haciendo uso de estructuras de control alternativas se verifican los posibles casos en los cuales los valores inapropiados de los parámetros que representan al día, mes y año de la fecha a crear, puede generar objetos fuera del dominio `Fecha` y para ellos se lanza una excepción. Esta excepción contiene un mensaje adecuado que indica cual fue la causa de error. Este mecanismo garantiza que de haber un error no se complete la ejecución de constructor (método o bloque de código donde se lance la excepción) y por ende no se cree ningún objeto fuera del dominio de objetos que denota la clase.

Pregunta: ¿Qué sucede al ejecutar las siguientes instrucciones?

```
{
    Fecha f = Fecha.Leer();

    Console.WriteLine(f.ToString());

}
```

para los siguientes juegos de datos.

a) `1/1/1980`

b) `30/2/2003`

c) `3.1/12/2003`

En el primer caso se imprime el mensaje `1/1/1980` que se corresponde con la `Fecha` correctamente tecleada.

En el segundo caso se detiene la ejecución del programa y se muestra un mensaje que comienza de la siguiente forma: `Excepción no controlada: System.Exception: Día incorrecto`

Finalmente en el tercer caso ni tan siquiera completar el tecleado de la tema, inmediatamente después de teclear el primer valor (3.1), se detiene la ejecución del programa y se muestra un mensaje que comienza de la siguiente forma: `Excepción no controlada: System.FormatException: La cadena de entrada no tiene el formato correcto`

Pregunta: ¿Qué tipo de filosofía para el tratamiento de errores se ha puesto de manifiesto?

Sin dudas que histeria pues se aborta la ejecución del programa .

Es significativo que en los dos casos que se ha cometido un error se ha detenido la ejecución del programa y se ha notificado con un mensaje de `Excepción no controlada ...` En el primer caso la terna tecleada no cumple con la pre-condición del constructor de la clase `Fecha` y este entonces levanta la excepción, mientras que en el segundo caso como valor del día se ha tecleado 3.1 que no es un número entero por lo que al llegar a la instrucción

```
uint d = uint.Parse(Console.ReadLine());
```

el motor de ejecución de .NET (CLR) se encarga de levantar una excepción (`FormatException`).

Es de destacar que en ambos casos se ha detectado un error y se ha levantado una excepción deteniendo la ejecución del programa (histeria). Sin embargo, una vez lanzada una excepción es posible escribir código que se ocupe de tratarla. Por defecto, si este código no se escribe la excepción provoca que la aplicación

aborte mostrando un mensaje de error en el que se describe la excepción producida (es lo que ha sucedido con anterioridad).

Pregunta: ¿Qué faltaría entonces para completar el mecanismo de excepciones basado en la filosofía *dispara* y *captura*?

De momento se ha garantizado que si no se cumplen las pre-condiciones se lanza una excepción al método que invocó al constructor y de esta forma se garantiza que el objeto no es creado. Luego, finalmente se necesita brindar un recurso al cliente para que pueda capturar esta excepción y darle tratamiento. Es decir, que le permita al programador realizar las acciones pertinentes que le permitan enmendar los errores detectados y que pueda continuar ejecutándose la aplicación en la medida de lo posible. De no escribirse código alguno se abortaría la aplicación mostrándose un mensaje que indica cual fue la causa y donde se ha producido.

Si se desea tratar la excepción hay que encerrar aquellas instrucciones que son posibles fuentes de errores en un bloque o instrucción de la siguiente forma:

```
try
    <instrucciones>
catch (<excepción1>)
    <tratamiento1>
catch (<excepción2>)
    <tratamiento2>
finally
      <instruccionesFinally>
```

El significado de esta nueva estructura sintáctica es el siguiente: si durante la ejecución de las instrucciones `<instrucciones>` se lanza una excepción del tipo `<excepción1>`, se ejecutan las instrucciones `<tratamiento1>`, análogamente si fuese de tipo `<excepción2>` y así sucesivamente hasta que encuentre una cláusula **`catch`** que pueda tratar la excepción producida. De no encontrarse ninguna que la trate entonces se elevaría la excepción al código desde el que se llamó al método que produjo la excepción (el método que contenía el bloque **`try`**/**`catch`**) y así sucesivamente hasta que se trate o se detenga la ejecución de la aplicación. En **`finally`** se colocan instrucciones que siempre se deseen ejecutar, se haya producido o no una excepción.

Sin embargo, debido a los objetivos del presente texto la sintaxis de la instrucción **`try`** se va a simplificar de la siguiente forma:

```
try
      <instrucciones>
catch (Exception e)
{
      Console.WriteLine(e.Message);
}
```

El significado de esta simplificación es que si se produce una excepción en las instrucciones del bloque **`try`** entonces se captura en el bloque **`catch`** y se muestra en mensaje de la misma.

Por el momento se ha simplificado la sintaxis de la instrucción **`try`** y de momento se utilizará solamente de esta forma ya que este es un mecanismo más complejo en el cual será posible profundizar cuando se aborde el tema relacionado con la Herencia que se sale de los marcos de este texto.

Ejemplo: Tratar las excepciones que se producen al intentar crear una instancia de `Fecha`.

Simplemente se debe encerrar en un bloque **`try`** la creación de la instancia y luego darle tratamiento a la excepción lanzada como se muestra a continuación:

```
{
    try
    {
          Fecha f = Fecha.Leer();
    }
    catch (Exception e)
    {
          Console.WriteLine(e.Message);
    }
}
```

Situación de análisis

Observe detenidamente el bloque de código que se presenta seguidamente y reflexione al respecto:

```
{
      Console.WriteLine("Antes de try ...");

      try
      {
            Fecha f = Fecha.Leer();

            Console.WriteLine(f.ToString());
      }
      catch (Exception e)
      {
            Console.WriteLine(e.Message);
      }
      Console.WriteLine("Después de try ...");
}
```

Pregunta: ¿Qué sucede al ejecutar el bloque de código anterior para los juegos de datos que se relacionan seguidamente?

a) `1/1/1980`

b) `30/2/2003`

c) `3.1/12/2003`

En el primer caso no se levanta ninguna excepción por lo que se imprimen los siguientes mensajes:

`Antes de try ...` // resultado de `Console.WriteLine("Antes de try ...");`

Mensajes para leer `día`, `mes` y `año` de f (1/1/1980) // resultados de `Fecha f = Fecha.Leer();`

`1/1/1980` // resultado de `Console.WriteLine(f.ToString());`

`Después de try ...` // resultado de `Console.WriteLine("Después de try ...");`

En el segundo caso se levanta una excepción en el constructor de `Fecha` porque para el mes 2 se ha tratado de asociarle 30 días, por lo que se imprimen los siguientes mensajes:

`Antes de try ...` // resultado de `Console.WriteLine("Antes de try ...");`

Mensajes para leer `día`, `mes` y `año` de f (30/2/2003)
// resultados de `Fecha f = Fecha.Leer();`

`Día incorrecto` // resultado de `Console.WriteLine(e.Message);`

```
Después de try ...
```
// resultado de `Console.WriteLine("Después de try ...");`

Note que en este caso no se ejecuta la instrucción

```
Console.WriteLine(f.ToString());
```

Esto está determinado porque la referida instrucción se encuentra en un bloque **`try`** después de una instrucción donde ha ocurrido una excepción.

Nota: De forma general, cuando se levanta una excepción en un determinado bloque la última instrucción en ejecutarse es la que se produce el lanzamiento de la instrucción.

Finalmente en el tercer caso se imprime el mensaje inicial

```
Antes de try ...
```
// resultado de `Console.WriteLine("Antes de try ...");`

Pero luego ni tan siquiera se logra completar el tecleado de la terna, porque inmediatamente después de teclear el primer valor (3.1) se levanta una excepción; es por ello que se imprimen los siguientes mensajes antes de concluir.

```
La cadena de entrada no tiene el formato correcto.
```
// resultado de `Console.WriteLine(e.Message);`

```
Después de try ...
```
// resultado de `Console.WriteLine("Después de try ...");`

II.3.4 Caso de Estudio: Integridad de la clase CuentaBancaria

Aún cuando se conoce que en algunos bancos los clientes pueden tener saldo negativo, se desea imponer la restricción para el dominio del problema (y por tanto para la clase `CuentaBancaria`), que el saldo tiene que ser positivo y por tanto el saldo mínimo también.

Situación de análisis

Ejecute las siguientes operaciones donde se utiliza la clase `CuentaBancaria`.

```
{
      CuentaBancaria.saldoMínimo = -2000;

      CuentaBancaria cuenta1 = CuentaBancaria.Leer();

      cuenta1.saldo = -50000;

      cuenta1.Extraer(-3000);

      Console.WriteLine(cuenta1.saldo);
}
```

Pregunta: ¿Qué valor se imprime luego de ejecutar las instrucciones anteriores?

`-2000`.

Pregunta: ¿A qué conclusión se puede llegar después de analizar las instrucciones anteriores?

Es posible asegurar que la clase `CuentaBancaria` es muy frágil porque no puede garantizar ni su misma integridad, qué pensar entonces de la integridad de sus instancias.

Para afirmar lo anterior se analiza a continuación cada una de las instrucciones presentadas con anterioridad:

- `CuentaBancaria.saldoMínimo = -2000;` en este caso es permisible que a la variable de clase `saldoMínimo` se le asigne un valor negativo, esto sucede porque a esta variable es posible acceder directamente.
- `CuentaBancaria cuenta1 = CuentaBancaria.Leer();` del capítulo anterior se conoce que en el método `Leer` de la clase `CuentaBancaria` se invoca al constructor de la

misma para crear una instancia. Sin embargo, este constructor no impone ninguna restricción para la asignación de valores al saldo. Por esta razón es permisible que se le asigne un valor negativo y entonces sería creada una instancia que no pertenece al dominio del problema.

- `cuenta1.saldo = -50000;` se permite el acceso directamente a la variable `saldo`, esto posibilita entonces que se le pueda asignar cualquier valor a la misma.
- `cuenta1.Extraer(-3000);` en este método, como no se restringe que el valor a extraer tiene que ser positivo, es permisible realizar la operación porque

  `–5000 – (–3000) = –2000`,

  entonces se cumple la condición `>= saldoMínimo` (2000); ahora el `saldo` de la instancia `cuenta1` es igual a `-2000` (valor que finalmente se imprime a través de (`Console.WriteLine(cuenta1.saldo);`), lo que implica que aún la instancia `cuenta1` no pertenece al dominio del problema. Incluso, aún cuando no se presenta ningún llamado a `Depositar`, en este hay aun más problemas que en `Extraer`.

Pregunta: ¿Cuáles serían las acciones a desarrollar para refinar la clase `CuentaBancaria` de forma tal que no se presenten las situaciones anteriores?

Las acciones deben estar encaminadas a restringir el acceso a la variables `saldoMínimo` y `saldo`, así como a refinar el constructor y los métodos `Depositar` y `Extraer`.

En el refinamiento del constructor se tiene que verificar que el parámetro para inicializar la variable `saldo` tenga valor mayor o igual que `saldoMínimo` como se muestra a continuación.

```
public CuentaBancaria(Persona unaPersona, double unSaldo)
{
      propietario = unaPersona;

      if (unSaldo >= saldoMínimo)
            saldo = unSaldo;
      else
            throw new Exception("saldo menor que saldo mínimo");

      uint día = (uint)System.DateTime.Now.Day;
      uint mes = (uint)System.DateTime.Now.Month;
      uint año = (uint)System.DateTime.Now.Year;
      fechaCreación = new Fecha(día, mes, año);
}
```

En el refinamiento del método `Depositar` solamente se tiene que verificar que el valor a depositar sea mayor que cero. En el caso del método `Extraer` es necesario verificar esto mismo para el valor a extraer y determinar si luego de restar dicho valor al saldo, este último es mayor o igual que el saldo mínimo. Seguidamente se muestra una variante de implementación de ambos métodos.

```
public void Depositar(double valor)
{
      if (valor > 0)
            saldo += valor;
      else
            throw new Exception("valor a depositar negativo o cero");
}
public void Extraer(double valor)
{
      if (valor <= 0)
            throw new Exception("valor a extraer negativo o cero");
      else
            if (saldo - valor >= saldoMínimo)
```

```
                saldo -= valor;
          else
                throw new Exception("saldo menor que saldo mínimo");
}
```

Particularmente en el caso de la variable `saldo`, una variante es hacerla privada y luego definir una propiedad de solo lectura (`Saldo`), las modificaciones a esta variable solamente serían permisibles a través de los respectivos métodos `Depositar` y `Extraer`.

En el caso de la variable `saldoMínimo`, se debe permitir que se le asignen valores, pero no directamente como hasta el momento. Para ello una vez más se utilizan las propiedades (por ahora se habían definido apenas propiedades de sólo lectura) pues también a una propiedad se le puede asignar la responsabilidad de escribir en una variable como se verá seguidamente (cláusula **`set`**).

```
public class CuentaBancaria
{
      public Persona propietario;
      double saldo;
      static double saldoMínimo;
      public Fecha fechaCreación;

      public double Saldo
      {
            get
            {
                  return saldo;
            }
      }

      public static double SaldoMínimo
      {
            get
            {
                  return saldoMínimo;
            }

            set
            {
                  if (value >= 0)
                        saldoMínimo = value;
                  else
                        throw new Exception("saldo mínimo negativo");
            }
      }
}
```

II.3.4.1 Especificación y uso de propiedades

En la sección II.3.2 se presentó el concepto de propiedad y se utilizó solamente para definir propiedades de solo lectura, mientras que en II.3.3 se realizó un uso más profundo de las mismas que es necesario formalizar para lo cual se ha diseñado esta sección.

Vale la pena recordar que para definir una propiedad se usa la siguiente sintaxis:

```
[<modificadores>] <tipoPropiedad> <nombrePropiedad>
{
      get
      {
            <códigoLectura>
```

```
        }

        set
        {
                <códigoEscritura>
        }
}
```

A continuación se utiliza la propiedad `SaldoMínimo` para ejemplificar cada una de las partes de la sintaxis de las propiedades.

```
<modificadores> public static
<tipoPropiedad> double

<nombrePropiedad> SaldoMínimo

<códigoLectura>
        return saldoMínimo;

<códigoEscritura>
        if (value >= 0)
                saldoMínimo = value;
        else
                throw new Exception("saldo mínimo negativo");
```

La forma de acceder a una propiedad, ya sea en modo lectura o escritura, es exactamente la misma que se utiliza para aceder a un campo de su mismo tipo.

Como ejemplo de lectura es posible ejemplificar la impresión de la propiedad `SaldoMínimo` de la clase `CuentaBancaria`:

```
Console.WriteLine(CuentaBancaria.SaldoMínimo);
```

Al ejecutarse esta instrucción se ejecuta el correspondiente `<códigoLectura>` donde se devuelve el valor de a variable de clase `saldoMínimo`.

Como ejemplo de escritura se tiene la modificación de la propiedad `SaldoMínimo` de la clase `CuentaBancaria`:

```
CuentaBancaria.SaldoMínimo = 100;
```

Al ejecutarse entonces esta instrucción se ejecuta el correspondiente `<códigoEscritura>` donde **`value`** se sustituye entonces por 100 y como es mayor que cero se le asigna a la variable de clase `saldoMínimo`.

Pregunta: ¿Qué sucede al ejecutarse la siguiente instrucción?

```
CuentaBancaria.SaldoMínimo = -2000;
```

Se levanta una excepción que contiene el mensaje: `saldo mínimo negativo`.

II.3.5 Ejercicios

1. Dada la siguiente definición de la clase automóvil situada en la BCL

```
class Automovil
{
 public string marca;
 public Fecha fechaFabricacion;
 public int cantidadRuedas, KmRecorridos;

 Automovil (string _marca, Fecha _fechaFab, int _nRuedas,
```

```
            int _KRmRecorridos)
 {
   marca =_marca;
   fechaFabricacion = _fechaFab;
   cantidadRuedas = _nRuedas;
   KmRecorridos = _KmRecorridos;
 }

 public Fecha FechaFabricacion
 {
    Set
    {
        fechaFabricacion = value;
    }
 }
}
```

a) En el siguiente listado:

```
{
 Automovil a1 = new Automovil (”Lada”, new Fecha(12,1,1980),4, 20);
 a1.marca = ” VW”;
   Console.writeln(a1.FechaFabricacion);
 Automovil a2 = new Automovil (”Chevy”, new Fecha(12,1,1870),-5, 30);
}
```

¿Qué errores se han cometido? Justifique.

b) ¿es esta una clase integra? Justifique.

c) Proponga cambios de forma tal que en todo momento se mantenga la integridad de los objetos. Comente variantes al respecto.

d) Proponga cuales implementaciones de propiedades seria aconsejable definir.

2. Diseñe e implemente una clase GuardiaSeguridad que permita representar a los agentes de vigilancia, de los cuales se conoce el nombre, fecha nacimiento, si paso el servicio militar y su aptitud para el trabajo(representada por un numero en el intervalo [1..5]), área donde cuida. Garantice al menos las responsabilidades que permitan:

a) Permitir el acceso o no a un trabajador al área donde se esta cuidando. Auxiliase de la clase trabajador implementada en II.1.

Nota: El trabajador podrá acceder si su área de trabajo coincide con la que esta siendo cuidada por el guardia.

b) Conocer en un momento dado a que cantidad de trabajadores se les ha negado el acceso

Nota: Recuerde que solo se podrá ser guardia si cumple con las siguientes condiciones:

a) Es mayor de 21 años.

b) Se ha cumplido con el servicio militar

c) Se posee una aptitud superior a 3.

En caso de incumplimiento de alguna de estas condiciones, se debe informar la causa que lo inválido. Añada a la solución del ejercicio el uso de propiedades según lo entienda.

3. Modifique la clase Triángulo, de forma tal que solo se permita crear instancias de la misma, solo si se cumple con la desigualdad triangular. Realice los referidos cambios para las 2 variantes de la clase que se han discutido anteriormente. Permita además que:

a) Luego de ser creadas las instancias, estas conserven su integridad en todo momento.

b) El acceso a las responsabilidades sea a través del uso de propiedades, ya sea para escritura o lectura.

4. Modifique la clase Persona vista en el capítulo I.2 de forma tal que el campo apellido sea sustituido por dos campos que expresen los apellidos que se heredan por parte paterna y materna. Permita además conocer dado los datos de otra persona que sea miembro de su familia, cuál es la relación que existe entre ellos (Padre, Madre o Hermano/a). Utilice las propiedades como método de acceso a los campos de los objetos.

5. Redefina las clases diseñadas en capítulos anteriores de forma tal que se aplique los conceptos de integridad y encapsulamiento. Garantice que el acceso a las responsabilidades de los objetos sea a través de propiedades.

6. Responda verdadero o falso según corresponda, argumente todas sus respuestas.

a) ¿Puede tener sentido una clase con todos sus constructores privados?

b) ¿Qué diferencias existen entre campo y propiedad?

c) ¿Qué mecanismos conoce para mantener los objetos íntegros en todo momento?

d) ¿Qué son las excepciones? Ponga un ejemplo de su uso.

e) ¿Qué variante usted propondría para sus aplicaciones a la hora del tratamiento de errores? ¿*Histeria* o *retorno silencioso*?

II.3.6 Bibliografía complementaria

- Capítulos 5, 7, 9 y 16 de Jose A. González Seco, El lenguaje de programación C#, 2002. Puede descargarse en http://www.josanguapo.com/

Tema III

III. ITERACION. ARREGLOS

Objetivos

- **Caracterizar** el concepto de secuencia.
- **Implementar** secuencias a través de arreglos en C#.
- **Definir** clases que contengan arreglos uni y bidimensionales.
- **Utilizar** estructuras de control repetitivas.
- **Obtener** algoritmos básicos.
- **Implementar** algoritmos básicos.
- **Obtener** aplicaciones que involucren instancias de las clases definidas en el desarrollo del tema.

Contenido

III.1 Arreglos unidimensionales. Estructuras de control repetitivas
III.2. Arreglos bidimensionales.

Descripción

El objetivo integrador de este tema es obtener aplicaciones en modo consola que involucren instancias de clases que precisen de arreglos.

El tema comienza con la presentación de situaciones de análisis que necesitan del concepto de secuencia para su solución, esto permitirá definir clases que contienen arreglos y a través de estas luego se presentan las estructuras de control iterativas y múltiples algoritmos.

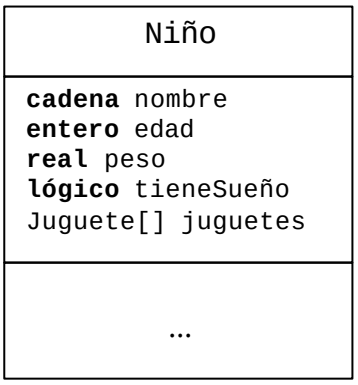
En la segunda parte del tema también se muestran situaciones de análisis pero que ahora requieren del concepto de tabla para su solución, esto permitirá definir clases que contengan arreglos bidimensionales y a través de estas es posible continuar ejercitando las estructuras de control iterativas y obtener múltiples algoritmos.

Capítulo III.1

III.1 Arreglos unidimensionales. Estructuras de control repetitivas

Los tipos de dato que se han utilizado hasta ahora, tan sólo pueden contener un dato(a la vez), ya sea un número, una cadena de caracteres o lógico; los mismos son, en cierto sentido, indivisibles. Esto quiere decir que las variables que representan valores de estos tipos de dato pueden almacenar un único objeto. Se tienen en cuenta incluso, los objetos que a su vez contienen otros objetos ya que conceptualmente el objeto *contenedor* es indivisible.

Pregunta: ¿Es posible implementar la clase `Niño` desarrollada en II.1, cuyo diagrama se presenta a continuación, con los recursos presentados hasta el momento?



El problema en este caso se presenta con el campo `juguetes`, hasta el momento se han implementado relaciones de uso donde un campo es una instancia de una clase. Sin embargo, ahora nos encontramos con un campo que contiene varias instancias de la referida clase, incluso el número de instancias es en principio indeterminado. En otras palabras, el campo `juguetes` contiene una *secuencia* de instancias de la clase `Juguete`.

"Una secuencia es un conjunto de valores de cualquier tipo, siendo significativo el orden (posición en la secuencia) entre ellos. La longitud de una secuencia, *long*, es el número de elementos que la componen. Por ejemplo la secuencia $S = s_1, s_2,..., s_n$, tiene n elementos. La secuencia vacía es la que tiene cero elementos, *long = 0*" [9]. En la mayoría de los lenguajes la forma de representar las secuencias es a través de arreglos (*arrays*)

Los arreglos se utilizan cuando se necesita almacenar múltiples valores del mismo tipo y con información relacionada.

III.1.1 Noción de secuencia. Arreglos unidimensionales en C#

"Un arreglo unidimensional es un tipo especial de variable que es capaz de almacenar en su interior y de manera ordenada varios elementos de un determinado tipo" [12]. Se entiende en este caso por orden, que cada elemento ocupa una posición, es decir, cada uno tiene predecesor (exceptuando el primero) y sucesor (exceptuando el último).

Seguidamente se muestran algunas formas de declarar e inicializar arreglos para representar secuencias en C#.

```
int[] temperaturas = {28, -5};
uint[] notas = {5, 4, 3, 4, 5};
double[] pesos = {70.3, 60, 65.7};
char[] palabra = {'H', 'o', 'l', 'a'};
```

```
string[] nombres = {"Ana"};
```

De forma general para declarar un arreglo en C#, se colocan corchetes vacíos entre el tipo y el nombre de la variable como se muestra a continuación:

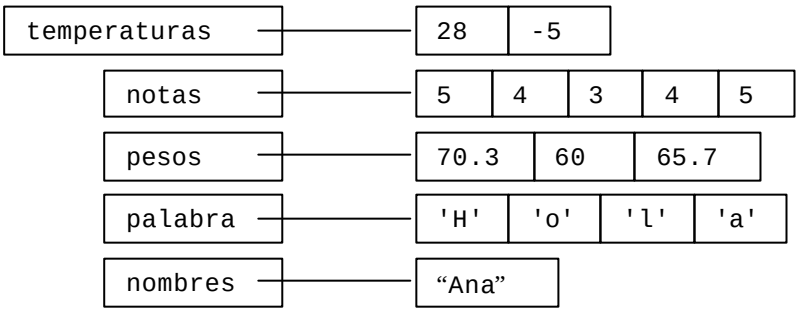
```
<identificador de tipo> [ ] <identificador de variable>;
```

Nota: Vale destacar que en este caso la combinación de símbolos "[ ]" no se está usando para representar opcionalidad sino como un elemento intrínseco de la declaración de los arreglos.

En C# los arrays son objetos cuya clase base es `System.Array`. Por lo tanto aunque la sintaxis para definir un `array` parezca similar a la de otros lenguajes, realmente estamos instanciando una clase .NET, lo que significa que todos los *arrays* declarados tienen los mismos métodos de la clase `System.Array`.

Al igual que el resto de las variables, los arreglos en C# se pueden instanciar en el momento de la declaración o posteriormente y que esta declaración (aún cuando utiliza semántica por referencia) no es obligatoriamente a través del operador **`new`** como se puede apreciar en los ejemplos mostrados con anterioridad.

Los arreglos utilizan semántica por referencia, las variables de arreglo guardan solamente la referencia al bloque de memoria donde realmente se encuentran los valores. En la figura siguiente se muestra una representación en memoria de los arreglos definidos e inicializados en el listado anterior.



Pregunta: ¿Cómo acceder a los distintos elementos que están almacenados en las variables definidas e inicializadas anteriormente.?

Para acceder a los elementos de un arreglo se utilizan índices como se muestra a continuación:

```
temperaturas[1]; notas[4]; pesos[2]; palabra[3]; nombres[0];
```

en todos los casos se ha accedido al último elemento de los respectivos arreglos, puesto que el índice de los arreglos en C# comienza en cero (0), esto significa que el primer elemento de un arreglo tendrá siempre índice cero (0), mientras que el último elemento tiene como índice la longitud del mismo (*long*, cantidad de elementos) menos uno (*long*-1).

Pregunta: ¿Cómo determinar el número de elementos que contiene el arreglo `pesos`?

Para determinar el número de elementos de cualquier arreglo se utiliza la propiedad `Length` que poseen todas las variables de arreglo, en lo particular para `pesos` sería de la siguiente forma:

```
pesos.Length
```

Situación de análisis

Sin dudas que no siempre es posible tener previamente definida la secuencia de valores para definir e inicializar el arreglo de una vez.

Pregunta: ¿Cómo es posible entonces utilizar los arreglos de forma dinámica?

El número de elementos del arreglo se indica en el momento de instanciarlo, esto puede ocurrir en la misma declaración (ejemplos anteriores) o posteriormente a través del operador **`new`**. El operador **`new`** se usa seguido del tipo y entre corchetes el número de elementos del arreglo a crear.

```
new <tipo de dato> [<cantidad de elementos>]
```

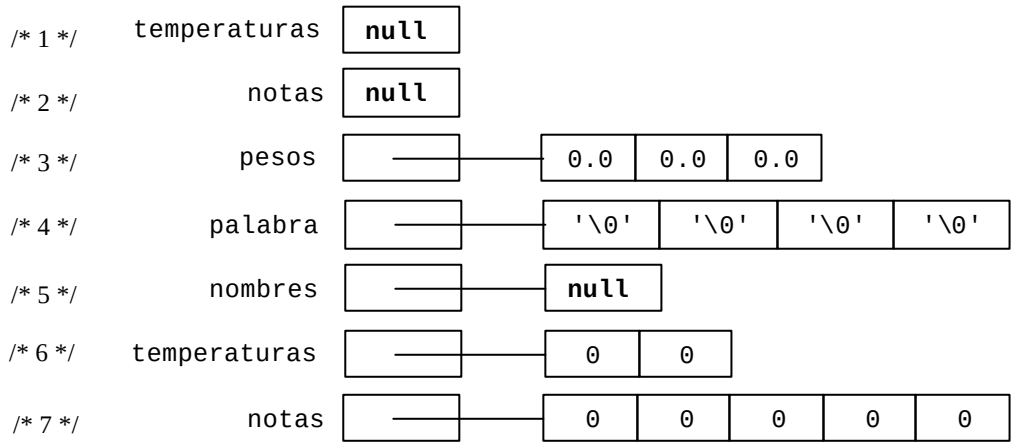
En ese momento se asigna el bloque de memoria necesario para crear el arreglo, guardando la dirección en una variable en caso de realizar la asignación.

Seguidamente se muestran algunos ejemplos de creación de arreglos a través del operador **new**.

```
{
    /* 1 */ int[] temperaturas;

    /* 2 */ uint[] notas;

    /* 3 */ double[] pesos = new double[3];

    /* 4 */ char[] palabra = new char[4];

    /* 5 */ string[] nombres = new string[1];

    /* 6 */ temperaturas = new int[2];

    /* 7 */ notas = new uint[5];
}
```

Pregunta: ¿Qué sucederá en la memoria al ejecutarse cada una de las instrucciones anteriores?

En el caso de las dos primeras instrucciones como solamente se están declarando las variables, se le asigna a ambas el valor por defecto de las referencias (**null**). Posteriormente, ante cada aplicación del operador **new**, se crea en la memoria (*heap*) una instancia para cada uno de los arreglos y la referencia es almacenada en las respectivas variables. Las diferentes localizaciones de los arreglos se inicializan en los valores por defecto de los respectivos tipos de datos. Observe una representación gráfica a continuación.



Sea cual sea la forma de instanciar el arreglo, luego de instanciarlo, se puede acceder a cualquiera de sus elementos. No hay que olvidar que la variable declarada como arreglo lo que contiene es una referencia al arreglo y no el arreglo en si mismo. Esa referencia es posible usarla para acceder individualmente a cualquiera de los elementos, disponiendo del índice del elemento al cual se desea acceder como ya se vio con anterioridad.

Como se ha expuesto con anterioridad, C# permite indicar el tamaño del arreglo de forma dinámica. Es por ello que se pueden declarar e instanciar los arreglos en tiempo de ejecución. Cuando se declara un arreglo y no se instancia, C# crea una referencia nula al arreglo como se demuestra en la representación gráfica de las dos primeras instrucciones en la figura anterior. Cualquier intento de acceso usando esta referencia causaría una excepción. Es posible usar la constante **null** para comprobar si se ha instanciado la variable o no como se muestra a continuación:

```
if (notas == null) …
```

Aún cuando existen diferencias en la implementación de los arreglos en los diferentes lenguajes, los arreglos son un recurso de construcción de tipos que se caracteriza porque asocia a un conjunto de valores índices un conjunto de valores que son las componentes. El nombre de la estructura establece una cierta función donde el dominio es el tipo que sirve de índice y la imagen es el tipo del componente [8]:

```
temperaturas : 0..4 ——— int

notas : 0..1 ——— uint

pesos : 0..2——— double

palabra : 0..3 ——— char

nombres : 0..0 ——— string
```

Para acceder a cualquiera de los elementos de los arreglos se realiza de la forma conocida a través de los corchetes y los índices.

Situación de análisis

Imagine ahora que fue creado el arreglo `notas` para cinco (5) asignaturas y luego tenemos una sexta (6) nota que se debe incluir.

Pregunta: ¿Cómo es posible adicionar una sexta nota si solamente se tienen cinco localizaciones en el arreglo `notas`? Manteniendo la referencia a través de la variable `notas` y sin perder los datos anteriores?

Esto no es posible directamente, es decir, los arreglos se crean dinámicamente, pero una vez creados son estáticos, no pueden cambiar su tamaño. Pero existe una solución para esto creando un arreglo *temporal* (`tempNotas`) con las nuevas capacidades que necesitamos (6), después se copian los elementos existentes en el arreglo original hacia este temporal y finalmente se cambia la referencia del original hacia el nuevo arreglo creado. Ahora se dispone entonces de una nueva localización creada para almacenar el nuevo elemento (la sexta nota). Seguidamente se muestra un bloque de código donde se ejemplifican estos pasos.

```
{
      uint[] tempNotas = new uint[6];

      notas.CopyTo(tempNotas, 0);

      notas = tempNotas;

      notas[5] = 4;
}
```

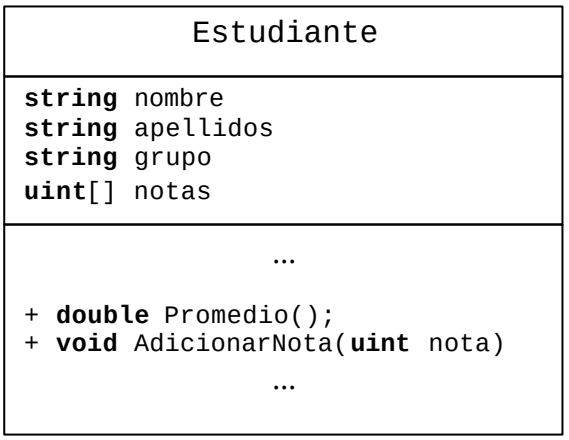
Cuando se asigna una variable que contiene un valor de un tipo de dato simple a otra variable del mismo tipo, lo que se hace es copiar el valor de la primera a la segunda (semántica de copia por valor) y por tanto se tienen dos variables con el mismo valor que son independientes entre si. Sin embargo, los arreglos utilizan semántica de copia por referencia, o sea, al asignar una variable de tipo arreglo a otra de su mismo tipo, lo que se hace es duplicar la referencia. Es decir, ahora se cuenta con un único arreglo pero con dos referencias al mismo en dos variables diferentes.

III.1.1.1 Caso de estudio: clase `Estudiante`, campo `notas`

Situación de análisis

Para desarrollar esta sección es conveniente simplificar el dominio del problema en el caso de estudio `Universidad` desarrollado en el capítulo I.1. La simplificación se propone específicamente en la clase `Estudiante` y en este caso radica en almacenar apenas el nombre, apellidos y solamente el valor de las

notas de las asignaturas aprobadas, así como la identificación del grupo al que pertenece, el promedio y un método que permita ir adicionando las respectivas notas. Seguidamente se muestra un diagrama de clases donde se representa el diseño de esta variante de la clase `Estudiante`:



Ejemplo: Codificar una variante de la clase `Estudiante` donde se implementen el constructor y el método `AdicionarNota`.

En una primera variante se asume que al crearse las instancias de la clase `Estudiante` no se tiene ninguna nota y que el arreglo `notas` va creciendo de uno en uno a través de `AdicionarNota` como se muestra en el listado siguiente.

```
public class Estudiante
{
    string nombre;
    string apellidos;
    string grupo;
    uint[] notas = new uint[0];

    public Estudiante(string nombre, string apellidos, string grupo)
    {
          this.nombre = nombre;
          this.apellidos = apellidos;
          this.grupo = grupo;
    }

    public void AdicionarNota(uint nota)
    {

          uint[] tempNotas = new uint[notas.Length+1];

          notas.CopyTo(tempNotas, 0);

          notas = tempNotas;

          notas[notas.Length - 1] = nota;
    }
}
```

Situación de análisis

Analice el segmento de código siguiente

```
{
      ...
      Estudiante e = new Estudiante(...);

      e.AdicionarNota(5);
      ...
}
```

Pregunta: ¿Qué sucede si mantenemos el mismo constructor y no inicializamos el campo `notas`?

Al ejecutarse el método `AdicionarNota` se aborta la ejecución del programa y se levanta una excepción con el siguiente mensaje:

```
Excepción no controlada: System.NullReferenceException: Referencia a
objeto no establecida como instancia de un objeto.
```

Esto se produce porque en el método `AdicionarNota` se intenta acceder a la propiedad `Length` :e la variable notas y esta nos ha sido instancia.

Pregunta: ¿Cómo evitar la problemática anterior?

En primer lugar podría controlarse la excepción y evitar que la aplicación aborte como se muestra a continuación:

```
{
    ...
    try
    {
          Estudiante e = new Estudiante(...);
          e.AdicionarNota(5);
    }
    catch (Exception exc)
    {
          Console.WriteLine(exc.Message);
    }
    ...
}
```

Sin embargo, esta solución prácticamente no resuelve nada porque los problemas graves están en la clase `Estudiante` que de esta forma no permite la inserción de ninguna nota.

En una segunda variante de solución podría instanciarse la variable `notas` en el constructor añadiendo la siguiente instrucción:

```
notas = new uint[0];
```

siendo esta solución equivalente a la inicial.

Si se insiste en no modificar el constructor, entonces se necesita realizar un análisis de casos con un subdominio para la adición de la primera nota, en cuyo caso habría que instanciar el arreglo notas y otro subdominio para el resto de las notas en cuyo caso se realizan acciones similares a las definidas en la solución original. Seguidamente se muestra un listado con esta variante de implementación del método `AdicionarNota`:

```
public void AdicionarNota(uint nota)
{
    if (notas == null)
```

```
            notas = new uint[1];
        else
        {
            uint[] tempNotas = new uint[notas.Length+1];
            notas.CopyTo(tempNotas, 0);
            notas = tempNotas;
        }

        notas[notas.Length - 1] = nota;
    }
```

Al término de esta sección podemos concluir que una secuencia se puede definir de forma recurrente (IV.1.1) [9]:

a. La secuencia vacía es una secuencia (`{}`).

b. Una secuencia `S` unida a un elemento `e` por la derecha es otra secuencia (`S.e`).

La longitud de una secuencia también se puede definir de forma recurrente:

c. La longitud de secuencia vacía es 0 (cero):.

d. La longitud de una secuencia unida a un elemento `e` por la derecha es igual a la longitud de la secuencia más 1: `(S.e).long == S.long + 1`.

III.1.2 Composición iterativa

Pregunta: ¿Cómo implementar el método `Promedio()` con los recursos que se dispone?

Una propuesta de algoritmo sería inicializar una variable temporal `suma` en cero para acumular el resultado de la sumatoria de las notas para luego dividir esta suma por `notas.Length`. No obstante, para obtener la sumatoria de las notas también tenemos que desarrollar un conjunto de acciones como se muestra a continuación de forma informal:

```
public double Promedio()
{
    double suma = 0;
    int i = 0;
    if (i < notas.Length)
    {
        suma += notas[i];
        i++;
        if (i < notas.Length)
        {
            suma += notas[i];
            i++;
            if (i < notas.Length)
            {
                suma += notas[i];
                i++;
                ...
            }
    }
    return suma / notas.Length;
}
```

Situación de análisis

Es significativo que no es posible realizar una definición formal del algoritmo anterior porque no se conoce a priori el número de elementos del arreglo y por tanto no es posible determinar el numero de **`if`** anidados que

se deberían implementar. Es decir, las técnicas conocidas para la descomposición de los algoritmos: secuenciación y análisis de casos, no son suficientes.

Por tanto, surge la necesidad de un nuevo mecanismo que permita tratar todos los elementos de una secuencia, o una parte de ellos en dependencia de que se cumpla cierta condición. Seguidamente se analizan construcciones de algoritmos iterativos modelando el problema como el tratamiento de una secuencia. El problema anterior se puede ver como el recorrido de una secuencia, en este caso una secuencia de valores enteros.

Otra forma de justificar la necesidad de una nueva composición es fijarse en el hecho de que las computadoras son máquinas de gran utilidad porque son capaces de realizar tareas muy laboriosas que implican millones de operaciones, de forma muy rápida. Si solamente se contase con la descomposición secuencial y el análisis de casos , los programas deberían ser extremadamente largos para expresar tareas que exijan gran cantidad de operaciones. Es por ello que se dispone de la composición iterativa como medio de escribir programas cortos cuya ejecución origine un gran número de operaciones.

Pregunta: ¿Cómo implementar entonces el método `Promedio()`?

Haciendo uso de una composición iterativa, una variante de algoritmo para el cálculo del promedio se muestra a continuación:

a. Inicializar una variable `double suma = 0;`

b. Recorrer el arreglo `notas` desde la posición `i = 0` hasta `i = notas.Length -1` y en cada paso realizar las siguientes acciones:

```
suma += notas[i];
i++;
```

Retornar `suma / notas.Length;`

Las variables `suma` e `i` tienen las mismas funciones que antes. La ejecución del algoritmo provocará que estas variables vayan cambiando de estado, evolucionando hacia el estado final o postcondición. De esta forma las variables `suma` e `i` comienzan en un estado inicial e_0 en el que su valor será cero; a continuación si al menos existe una nota el estado de estas variables cambia pasando al estado e_1 (`suma == notas[0]` e `i == 1`); a continuación, si al menos existen dos notas vuelve a cambiar el estado de estas variables pasando ahora a e_2 (`suma == notas[0] + notas[1]` e `i == 2`); y así sucesivamente hasta llegar al estado final e_f en el que la variable suma debe contener la sumatoria de todas las notas y la variable `i` el valor `notas.Length`.

En general, es posible ver una iteración como la repetición de un mismo tratamiento para cada elemento de la secuencia. En el algoritmo del promedio, a una secuencia de enteros se aplica como tratamiento elemental las dos asignaciones `suma += notas[i]` e `i++`. Después de tratar cada elemento de la secuencia, se alcanza un estado intermedio. El último estado de la iteración cumplirá las postcondición.

Cada estado intermedio está determinado por el valor de las variables que son modificadas en cada tratamiento elemental. Existe una relación entre el valor de las variables y los elementos de la secuencia ya tratados, que expresa que el tratamiento ha sido aplicado a todos esos elementos [9]. En el ejemplo anterior, en cada estado intermedio los valores de las variables `suma` e `i` reflejan que se han sumado `i++` notas cuya sumatoria es `suma`. Esto quiere decir que en cada estado intermedio se cumple la siguiente condición: *se conoce la solución para la parte izquierda de la secuencia tratada*. Esta relación se denomina *invariante*, y es posible afirmar que el invariante es una condición que se cumple en cada estado intermedio, al principio y al final de cada iteración [9].

Al diseñar algoritmo iterativos, el objetivo es conseguir la postcondición, esto es, que al final de la iteración los valores de las variables contengan la solución del problema. Puesto que en cada estado intermedio se cumple el invariante, también se cumplirá en el estado final [9].

Una heurística para diseñar una iteración que implica el recorrido de todos los elementos de una secuencia, es aplicar un razonamiento inductivo basado en la idea de invariante. Supuesto una secuencia S

que contiene `n` elementos, `n >= 0`, y sea `P` la postcondicion que se desea alcanzar, es posible realizar el siguiente razonamiento: sea `p` la variable que almacenará el resultado (en realidad, puede ser necesario más de una variable), y supuesto que nos encontramos en el estado `i – 1, 1 <= i <= n`, (elemento actual S_i), entonces `p` tendrá un valor tal que se cumpla el invariante, entonces, ¿qué tratamiento se debe aplicar sobre S_i tal que en el siguiente estado se siga cumpliendo el invariante? Obviamente, hay que definir el valor a asignar a `p` al inicio del proceso, antes de comenzar el tratamiento de la secuencia [9].

III.1.3 Estructuras de control iterativas

Los algoritmos que manejan secuencias se pueden clasificar en dos grupos: algoritmos de *recorrido* y algoritmos de *búsqueda*. En los primero se recorre toda la secuencia y en los segundos se recorre la secuencia hasta encontrar un elemento que cumpla cierta propiedad.

Pregunta: ¿Cómo clasificar el algoritmo del ejemplo desarrollado con anterioridad para el cálculo del promedio?

Este es un clásico ejemplo de algoritmo de recorrido.

En la sección anterior se ha visto como de manera general para dar solución a ciertos problemas mediante algoritmos es necesario repetir un bloque de instrucciones un número determinado de veces seguidas, o como se verá más adelante mientras se satisfaga una condición determinada. Obviamente no se escribirá múltiples veces el código, sino que será creado un ciclo, bucle o *estructura de control repetitiva* (*iterativa*).

"Una de las características importantes que se pueden aprovechar de las computadoras es precisamente su capacidad de repetir la ejecución de secuencias de instrucciones a una gran velocidad y con alto grado de confiabilidad" [8]. Para estos fines, precisamente se definen en los lenguajes de programación las estructuras de control iterativas (en C# **`for`**, **`while`**, **`do while`** y **`foreach`**). El objetivo de estas estructuras es permitir la expresión de la repetición de una secuencia de instrucciones, a estas estructuras se les denomina *ciclo*, *lazo* o *bucle*.

III.1.3.1 Estructura de control `for`

Uno de los ciclos más conocidos y usados es el basado en la instrucción **`for`** y que suele estar controlado por un contador o variable de control y que tiene la siguiente sintaxis:

```
for (<instrucciones 1>; <expresión>; <instrucciones 2>)
      <instrucciones>
```

La semántica de esta instrucción es la siguiente:

`<instrucciones 1>;` se ejecutará una sola vez al inicio del ciclo, generalmente se realizan inicializaciones y declaraciones de variables puesto que como se dijo con anterioridad, esta solo se ejecuta una vez. En caso de que se quiera realizar o ejecutar mas de una instrucción en este momento, dichas instrucciones se deben separar por comas (",").

`<expresión>;` es evaluada en cada ciclo y en dependencia del valor que devuelva, dependerá que el bucle continúe ejecutándose (valor de la evaluación **`true`**) o no (**`false`**). Es válido destacar o advertir que de no colocarse nada en esta parte, el ciclo tomará como **`true`** el valor devuelto por lo que en principio se podrá repetir infinitamente.

`<instrucciones 2>;` es ejecutado siempre en cada ciclo al terminar de ejecutar todas las instrucciones que pertenecen al bucle **`for`** en cuestión. Por lo general puede contener alguna actualización para las variables de control. Análogamente a `< instrucciones 1>` en caso de querer ejecutar en este momento más de una instrucción se deben separar por comas.

Ejemplo: Implementar el algoritmo propuesto con anterioridad para el cálculo del promedio.

```
public double Promedio()
{
    double suma = 0;
    for (uint i = 0; i < notas.Length; i++)
```

```
            suma += notas[i];

        return suma / notas.Length;
    }
```

Situación de análisis

En esta variante de implementación el recorrido del arreglo `notas` es desde el primer elemento hasta el último.

Ejemplo: ¿Implementar el método promedio haciendo el recorrido inverso del arreglo `notas`?

```
    {
        double suma = 0;
        for (int i = notas.Length - 1; i >= 0; i--)
              suma += notas[i];

        return suma / notas.Length;
    }
```

III.1.3.2 Estructura de control `while`

Situación de análisis

Precisar en el dominio del problema de la clase `Estudiante` simplificada, se tiene que las notas son valores entre 2 y 5, correspondiéndose el valor 2 con la categoría de asignatura desaprobada.

Pregunta: ¿Cómo determinar si una instancia de la clase `Estudiante` tiene alguna asignatura desaprobada?

```
    public bool AsignaturaDesaprobada()
    {
        for (uint i = 0; i < notas.Length; i++)
              if (notas[i] == 2)
                    return true;

        return false;
    }
```

Otra variante:

```
    {
        uint i;

        for (i = 0; i < notas.Length && notas[i] != 2; i++);

        return i < notas.Length;
    }
```

A veces no es posible saber de antemano el número de veces que se va a repetir la ejecución de una posición de código mientras una cierta condición sea cierta. En realidad como se ha visto en esta sección, la instrucción **`for`** es un bucle condicional, aunque su uso más habitual está asociado con un contador. Para definir estos ciclos condicionales es posible utilizar la estructura de control **`while`** cuya sintaxis es la siguiente:

```
    while <condición>
          <instrucciones>
```

Esta estructura de control permite realizar un bucle (`<instrucciones>`) que se repetirá mientras la `<condición>` sea cierta, o dicho de otra manera, el bucle terminará cuando la `<condición>` sea falsa. Cabe señalar que con esta estructura la `<condición>` se chequea siempre al principio del bucle por lo que si la primera vez que se evalúa `<condición>`, es falsa, el ciclo no llegará nunca a ejecutarse.

A continuación se muestra una variante del método `AsignaturaDesaprobada` a través de la estructura de control **`while`**:

```
{
      uint i = 0;

      while ( i < notas.Length && notas[i] != 2)
          i++;

      return i < notas.Length;
}
```

Pregunta: ¿Cómo clasificar el algoritmo de implementación del método `AsignaturaDesaprobada`?

Este es un clásico ejemplo de algoritmo de búsqueda.

III.1.3.3 Estructura de control `do-while`

La estructura de control **`do-while`** es otra sentencia de iteración en la que la condición se evalúa por primera vez después de que las instrucciones del ciclo se hayan ejecutado. Esto quiere decir que las sentencias del bucle **`do-while`**, al contrario que las del ciclo **`while`**, al menos se ejecutan una vez.

La sintaxis de esta estructura es la siguiente:

```
do
  <instrucciones>
while <condición>;
```

Si se impone la precondición de que al menos siempre existe una nota, entonces el método `AsignaturaDesaprobada` podría implementarse a través de las siguientes instrucciones:

```
{
      uint i = 0;

      do
            i++;
      while ( i < notas.Length && notas[i] != 2);

      return i < notas.Length;
}
```

III.1.3.4 Estructura de control `foreach`

El ciclo **`foreach`** repite las instrucciones para cada elemento de un arreglo o colección. La finalidad de esta estructura es recorrer todos los elementos de un arreglo o colección, sin necesidad de índices ni valores mínimos o máximas. Sintácticamente, como se muestra seguidamente, tras la palabra y entre paréntesis se debe insertar una variable del tipo adecuado que va tomando el valor de cada uno de los elementos que existan en el arreglo, ejecutando el ciclo para cada uno de ellos.

```
foreach ([<tipo de dato>]<variable> in <arreglo o colección>)
      <instrucciones>
```

Ejemplo: Implementar variantes de los métodos `Promedio` y `AsignaturaDesaprobada` a través de la estructura de control **`foreach`**.

```
public double Promedio()
{
    double suma = 0;
    foreach (uint nota in notas)
          suma += nota;
```

```
    return suma / notas.Length;
}
public bool AsignaturaDesaprobada()
{
    foreach (uint nota in notas)
        if (nota == 2)
            return true;

    return false;
}
```

Utilizar esta última estructura de control en lugar del **`for`** tiene algunas ventajas. Una de ellas es que si después del diseño original del programa, este se modifica y el número de elementos del arreglo se reduce o amplia, puede ser que en un bucle **`for`** se tengan que retocar los índices, mientras que en un **`foreach`** esto no es necesario ya que se recorren los elementos que existan en ese mismo momento.

III.1.4 Otros algoritmos básicos

Ejemplo: Definir un algoritmo para determinar la cantidad de asignaturas desaprobadas en instancias de la clase `Estudiante`.

Este es clásico algoritmo de contar que podemos describir de la siguiente forma:

Inicializar las variables **`uint c = 0`** e `i = 0;`

Recorrer el arreglo `notas` desde la posición `i = 0` hasta `i = notas.Length -1` y en cada paso realizar las siguientes acciones:

```
if (notas[i] == 2)
    c++;
i++;
```

Retornar `c;`

Ejemplo: En el algoritmo definido con anterioridad, determinar el estado inicial, el invariante y la postcondición.

Estado inicial: **`uint`** `c = 0` e `i = 0;`
Invariante:
```
if (notas[i] == 2)
    c++;
i++;
```

Postcondición: variable `c` contiene la cantidad de asignaturas desaprobadas e `i = notas.Length;`

Ejemplo: Incluir un método en la clase `Estudiante` para determinar la cantidad de asignaturas desaprobadas.

```
public uint AsignaturasDesaprobadas()
{
    uint c = 0;

    for (uint i = 0; i < notas.Length; i++)
        if (notas[i] == 2)
            c++;

    return c;
}
```

Otra variante ahora con **`foreach`**:

```
{
    uint c = 0;
```

```
    foreach (uint nota in notas)
          if (nota == 2)
                c++;

    return c;
}
```

Ejemplo: Definir una expresión general para el algoritmo de contar los elementos de un arreglo (a) que cumplen una `<condición>`.

Este es clásico algoritmo de contar que podemos describir de la siguiente forma:

Inicializar la variables `uint c = 0;`

Para todos los elementos del arreglo a:

```
if <condición>
     c++;
```

Retornar `c;`

Ejemplo: Definir un algoritmo para determinar la mayor nota en instancias de la clase `Estudiante`.

Este es clásico algoritmo para la determinación del mayor (menor) podemos describir de la siguiente forma:

Inicializar las variable `uint mayorNota` con el menor valor del dominio de valores `uint (uint.MinValue)`

Para cada elemento ($nota_i$) del arreglo `notas`:

```
if mayorNota < notai
     mayorNota = notai
```

Retornar `mayorNota;`

Otra variante:

Inicializar las variable `uint mayorNota` con el primer valor del arreglo `notas`

Para cada elemento ($nota_i$) del arreglo `notas` exceptuando el primero:

```
if mayorNota < notai
     mayorNota = notai
```

Retornar `mayorNota;`

Ejemplo: Incluir un método en la clase `Estudiante` para determinar la mayor nota.

```
public uint MayorNota()
{
    uint mayorNota = uint.MinValue;

    for (uint i = 0; i < notas.Length; i++)
          if (mayorNota < notas[i])
                mayorNota = notas[i];

    return mayorNota;
}
```

Otras variantes:

```
{
    uint mayorNota = uint.MinValue;
```

```
    foreach (uint nota in notas)
          if (mayorNota < nota)
                mayorNota = nota;

    return mayorNota;
}
{
    uint mayorNota = notas[0];

    for (uint i = 1; i < notas.Length; i++)
          if (mayorNota < notas[i])
                mayorNota = notas[i];

    return mayorNota;
 }
```

III.1.5 Caso de estudio: algoritmo pares `Si == "0"` y `Sj == "1"`

Ejemplo: Sea una secuencia `S` formada por n caracteres, se desea obtener el número de pares (`i,j`) tales que `1 <= i < j <= n` y S_i `== "0"` y S_j `== "1"` (idea original y solución en [9]).

Dada la secuencia S = "10100110", el número de pares es 7, que son: {(2,3), (2,6), (2,7), (4,6), (4,7), (5,6), (5,7)}. Por tanto, el valor a calcular sería 7, no la lista de pares.

Como postcondición es posible establecer la siguiente:

`np ==` número de pares (`i,j`) tales que `1 <= i < j <= n, n == S.long` siendo S_i `== "0"` y S_j `== "1"`

Sustituyendo `S` por P_{iz} se obtiene el siguiente invariante

`np ==` número de pares (`i,j`) tales que `1 <= i < j <= k, k ==` P_{iz}`.long` siendo S_i `== "0"` y S_j `== "1"`

donde `k` representa el numero de elementos que ya han sido tratados. Por tanto el invariante establece que en cada estado intermedio, la variable `np` tiene asignado el número de pares en la P_{iz}. Ahora cabe hacerse la siguiente pregunta clave, ¿qué se debe hacer para mantener el invariante, por cada nuevo elemento tratado, supuesto que se cumplía el invariante para la secuencia ya tratada?

Como en cada estado intermedio se tiene el valor de `np` para la secuencia ya tratada, entonces el objetivo será encontrar como actualizar `np` cuando se trate un nuevo elemento. Ahora se considera como secuencia P_{iz}`.ea`, siendo `ea` el nuevo elemento que puede ser un "`0`" o un "`1`". Luego la respuesta se puede descomponer en dos casos, uno por cada posible valor del elemento. Si se trata de un "`0`", el valor de `np` no puede cambiar ya que el nuevo elemento no genera nuevos pares en relación a la P_{iz}. Sin embargo, si se trata de un "`1`" será necesario incrementar el valor de `np`. ¿En cuánto se tiene que incrementar `np`? Pues en el número de pares que tiene como extremo el actual, o sea en el número de elementos con valor cero que se encuentran en la parte izquierda (`nc`).

Prácticamente, ya se tienen todos los elementos para escribir el algoritmo que quedaría de la siguiente forma:

Inicialización: `nc = 0; np = 0;`

Cuerpo:

```
if si == "0"
    nc++
if (si == "1")
    pc += nc
```

Situación de análisis

En C#, las variables de tipo **`string`** pueden verse, en cierta medida, como arreglos. Por ejemplo, para acceder al i-ésimo carácter de una variable **`string`** se utiliza la misma sintaxis que para acceder al i-ésimo elemento de un arreglo.

Ejemplo: Escribir una aplicación consola que lea una cadena de caracteres (`s`) y permita obtener el número de pares (`i,j`) tales que `1 <= i < j <= n` tales que `s[i] == "0"` y `s[j] == "1"`.

Sin dudas que implementando el algoritmo desarrollado con anterioridad, se obtiene una solución al ejemplo como se muestra a continuación:

```
{
    uint np = 0, nc = 0;
    string sec = Console.ReadLine();
    for (uint i = 0; i < sec.Length; i++)
    {
          if (sec[i] == '0')
                nc++;
          if (sec[i] == '1')
                np += nc;
    }
    Console.WriteLine(np);
}
```

III.1.6 Caso de estudio: clase `Vector`

Hasta el momento se ha presentado el concepto de arreglos como objetos derivados de `Array`, éstos tienen que ser instanciados, utilizan semántica de referencia y se auxilian de un conjunto de métodos y propiedades heredados. Como se vio en las secciones anteriores y se verá en la presente, existen situaciones en las que el concepto de arreglo sirve como estructura para representar y resolver un problema determinado.

Situación de análisis

Se define como `Vector v`, que pertenece al espacio R^n, a una n-upla de la forma

$(a_1, a_2, \ldots, a_n)$,

donde para todo a_i se cumple que pertenece al conjunto de los números reales y `n` es mayor o igual que 0.

Se define además como dimensión de este `Vector v`, la cantidad de elementos que posee el mismo, luego si `v` tiene `n elmentos v.Dimensión` es igual a `n`.

Podemos entonces generalizar a todos los vectores de cualquier dimensión en una clase `Vector`, que además de la dimensión tenga las siguientes responsabilidades.

Sean `v1 =` $(a_1, a_2, \ldots, a_n)$ y `v2 =` $(b_1, b_2, \ldots, b_m)$ dos vectores, entonces si

`v1.Dimensión == v2.Dimensión` (`n == m`) se cumple que:

`v3 = v1 + v2`, donde `v3 =` $(a_1 + b_1, a_2 + b_2, \ldots, a_n + b_m)$

`v1 * v2`= sumatoria, es decir el producto escalar es una operación de

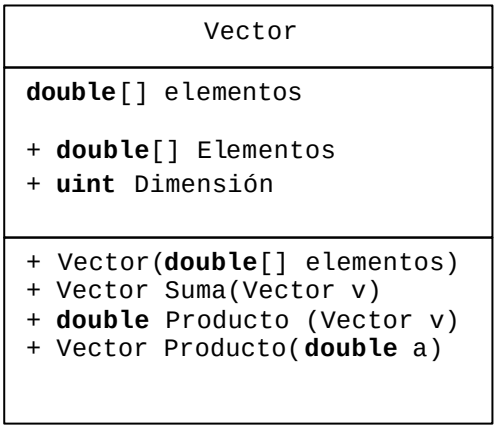
E x E R, siendo E el espacio de v_1 y v_2.

`v4 = v1 * a`, donde `v4 = (a1 * a, a2 * a, …,an * a)`

Ejemplo: Diseñar e implementar la clase `Vector` en C#.

En primer lugar esta clase tiene que contener un arreglo con los elementos de las instancias de `Vector` y luego al menos, responsabilidades para poder informa r sobre la dimensión y realizar las operaciones definidas con anterioridad.

De momento definiremos métodos para representar las operaciones aunque un poco más tarde veremos la posibilidad de utilizar operadores para las referidas operaciones.




```
public class Vector
{
    double[] elementos;

    public Vector(double[] elementos) ...

    public Vector Suma(Vector v) ...

    public double Producto (Vector v) ...

    public Vector Producto(double a) ...

    public uint Dimensión
    {
          get
          {
                return elementos.Length;
          }
    }

    public double[] Elementos
    {
          get
          {
                return elementos;
          }
    }
}
```

Pregunta: ¿Por qué usar una propiedad `Elementos` si lo único que hace es retornar la variable elementos?

Nótese que esta propiedad ha sido declarada de solo lectura. Entonces de esta forma, es decir, a través de la propiedad tenemos un nivel de seguridad sobre la variable. Al menos la dimensión de las instancias de `Vector` no pueden ser modificadas pues la referencia a la variable de arreglo elementos no puede ser

modificada. No sucede lo mismo con sus elementos componentes que pueden ser modificados arbitrariamente para lo cual debemos buscar una solución en el futuro.

Ejemplo: Implementar el constructor de la clase `Vector`.

Nótese que esta es una propuesta de constructor para crear un objeto `Vector` e inicializarlo con los valores provenientes de un arreglo que se pasa como parámetro. En este caso debemos garantizar que se realice una copia de los elementos de dicho arreglo a la variable elementos, pues si en su lugar asignamos el arreglo tendremos dos referencias y por lo tanto las modificaciones que se realicen en el arreglo que se ha pasado como parámetro tendrían incidencia directa en el objeto Vector creado. Ver listado III.1.3.1.

```
public class Vector
{
    ...

    public Vector(double[] elementos)
    {
          this.elementos = new double[elementos.Length];
          elementos.CopyTo(this.elementos, 0);
    }
    ...
}
```

Ejemplo: Implemente el método `Suma` de la clase `Vector`.

Dada la definición del operador suma (+) para vectores, entonces el método `Suma` tiene primero que verificar el cumplimiento de la pre-condición de que ambos vectores tengan la misma dimensión en caso positivo, luego retornar un tercer vector que contiene la respectiva suma.

```
public class Vector
{
    ...

    public Vector Suma(Vector v)
    {
          if (Dimensión != v.Dimensión)
                throw new Exception("Dimensiones diferentes");

          Vector vSuma = new Vector(Dimensión);
          for (uint i = 0; i < Dimensión; i++)
                vSuma [i] = elementos[i] + v.Elementos[i];

          return vSuma;
    }

...
}
```

Ejemplo: Implementar el método que implementa el producto escalar.

El algoritmo en este caso consiste en acumular la sumatoria del producto de las componentes que aparecen en la misma posición en ambos vectores.

```
public class Vector
{
    ...

    public double Producto(Vector v)
    {
          if (Dimensión != v.Dimensión)
                throw new Exception("Dimensiones diferentes");
```

```
            double producto = 0;
            for (uint i = 0; i < Dimensión; i++)
                  producto += elementos[i] * v.Elementos[i];

            return producto;
      }

...
}
```

Ejemplo: Implemente el método para obtener el producto escalar.

En este método tenemos que retornar un segundo `Vector` que se compone de los elementos del `Vector` que se está multiplicando por el escalar en cuestión.

```
public class Vector
{
    ...

    public Vector Producto(double a)
    {
          Vector vProducto = new Vector(Dimensión);
          for (uint i = 0; i < Dimensión; i++)
                vProducto [i] = elementos[i] * a;

          return vProducto;
    }
}
```

III.1.6.1 Indizadores (indexers)

Situación de análisis

Aunque no lo hemos hecho explícito, con esta definición de la clase `Vector` es evidente que cuando deseamos acceder a un elemento determinado de algún objeto de su dominio debemos hacerlo a través de su propiedad `Elementos` como lo vemos a continuación:

```
v.Elementos[i];
```

de hecho este es el recurso que utilizamos en el método `Suma`.

Evidentemente este acceso a los elementos está permeado por la implementación computacional. Esta permeabilidad está dada porque los accesos hasta el momento se están llevando a cabo a través de propiedades (o métodos de acceso). Por otra, no hay dudas que de forma natural a los elementos de las instancias de la clase Vector se accede directamente al i-ésimo elemento de la forma `v[i]`. Esto entra un poco en contradicción con el modelo de objetos donde se asume como una bondad el acercar lo más posible la implementación computacional a lo que sucede en la realidad.

"Sin embargo, para ayudar a desarrollar un lenguaje mejor y más intuitivo, el equipo de diseño del lenguaje C# observó esto y se preocupó: ¿por qué no tener la habilidad de tratar un objeto que es el un `array` como un `array`? De esta idea nación el concepto de indizadores (indexers) [13].

Un indizador es una definición de cómo se puede aplicar el operador de acceso a arreglos (`[]`) a los elementos de un tipo de dato. Esto es especialmente útil para hacer más clara la sintaxis de acceso a elementos de objetos que pueden contener colecciones de elementos, pues permite tratarlos como si fueran arreglos normales [12].

Coincidimos con Archer [13] cuando enuncia: Dado que a las propiedades se les conoce como campos o variables inteligentes, a los indizadores se les llama `arrays` inteligentes, y por ello tiene sentido que las propiedades y los indicadores compartan la misma sintaxis. De hecho definir indizadores es muy parecido a definir propiedades, con dos diferencias importantes: la primera, los indizadores toman un argumento índice;

la segunda, como la propia clase se está utilizando como un `array`, la palabra reservada **`this`** se utiliza como nombre del indizador. Seguidamente se muestra una aplicación de este interesante recurso en la clase `Vector`.

```
public class Vector
{
      ...
      public double this[uint i]
      {
            get
            {
                  if (i > Dimensión)
                        throw new Exception(“Indice fuera de rango”);

                  return elementos[i];
            }
      }

      public Vector Suma(Vector v)
      {
            if (Dimensión != v.Dimensión)
                  throw new Exception("Dimensiones diferentes");

            Vector vSuma = new Vector(Dimensión);
            for (uint i = 0; i < Dimensión; i++)
                  vSuma [i] = this[i] + v[i];

            return vSuma;
      }
}
```

Tenga en cuenta que, con independencia de cómo se almacenan los datos internamente (esto es como un `array`, colección, flujo, etc.), los indizadores son simplemente un medio para que el programador que instancia la clase escriba código como este:

```
{
      double[] elementosVector = {1, 2, 3};

      Vector v = new Vector(elementosVector);

      Console.WriteLine(“({0),{1),{2)”, v[0], v[1], v[2]);
}
```

“Sin embargo, como cualquier característica del lenguaje, los indizadores tienen un lugar. Debería utilizarse solo donde fuera intuitivo tratar un objeto como un array” [13].

Los indizadores permiten definir código a ejecutar cada vez que se acceda a un objeto del tipo que del que son miembros usando la sintaxis propia de los arreglos, ya sea para leer o para escribir. A diferencia de los arreglos, los índices que se pasan entre corchetes no tienen porqué ser enteros, pudiéndose definir varios indizadores en un mismo tipo siempre y cuando cada uno tome un número o tipo de índices diferentes [12].

III.1.6.2 Redefinición de operadores

Ejemplo: Implementar un segmento de código que imprima la suma entre los vectors

v1 = (1, 2, 3, 4, 5) y v2 = (6, 7, 8, 9, 10).

En este caso tenemos que crear dos instancias de la clase `Vector` para `v1` y `v2` respectivamente y luego imprimir el `Vector` resultante de la suma de estos como se muestra a continuación:

```
{
    double[] elementosV1 = {1, 2, 3, 4, 5};
    double[] elementosV2 = {6, 7, 8, 9, 10};

    Vector v1 = new Vector(elementosV1);
    Vector v2 = new Vector(elementosV2);

    Vector v3 = v1.Suma(v2);

    Console.WriteLine(v3.ToString());
}
```

Situación de análisis

Una vez más aparecen construcciones del lenguaje que no reflejan la forma natural del fenómeno real, es evidente que la construcción `v1 + v2` es más natural y hace más claro y legible el código que `v1.Suma(v2)`.

Vale la pena recordar que en I.2 se definió que un operador en C# no es más que un símbolo formado por uno o más caracteres que permite realizar determinada operación entre uno o más datos y produce un resultado.

En C# viene predefinido el comportamiento de sus operadores cuando se aplican a ciertos tipos de datos. Por ejemplo, si se aplica el operador + entre dos objetos **`int`** devuelve su suma, y si se aplica entre dos objetos **`string`** devuelve la concatenación. [12]

En C# también se permite que el programador pueda definir el significado de la mayoría de estos operadores cuando se apliquen a objetos de tipos que él haya definido, y esto es a lo que se le conoce como redefinición de operador o sobrecarga de operadores.

"La sobrecarga de operadores permite que los operadores de C# existentes se redefinan para que se puedan utilizar con tipos definidos por el usuario. La sobrecarga de operadores se ha llamado azúcar sintáctico, debido al hecho de que el operador sobrecargado es simplemente una forma de invocar a un método. También se ha dicho que este recurso no añade nada fundamental al lenguaje. Aunque esto es técnicamente cierto, la sobrecarga de operadores realmente ayuda en uno de los aspectos más importantes de la programación: la abstracción. Dicho de forma sencilla, la sobrecarga de operadores ayuda a la creación de software menos caro de escribir y mantener" [13]. En el listado siguiente se redefine el operador + para la clase `Vector`.

```
public class Vector
{
    public static Vector operator + (Vector v1, Vector v2)
    {
            return v1.Suma(v2);
    }

      ...
}
```

Nótese que en este caso hemos reutilizado el método Suma en la implementación implementación de la semántica del operador + de la clase Vector.

A través de este recurso del lenguaje podemos entonces realizar sumas de vectores de manera muy natural, clara y legible como se muestra en el listado III.1.1.2.

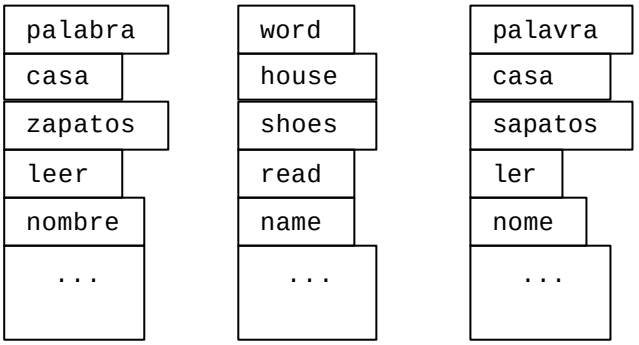
```
{
      double[] elementosV1 = {1, 2, 3, 4, 5};
      double[] elementosV2 = {6, 7, 8, 9, 10};

      Vector v1 = new Vector(elementosV1);
      Vector v2 = new Vector(elementosV2);

      Vector v3 = v1 + v2;

      Console.WriteLine(v3.ToString());
}
```

III.1.7 Caso de estudio: clase `Traductor de palabras`

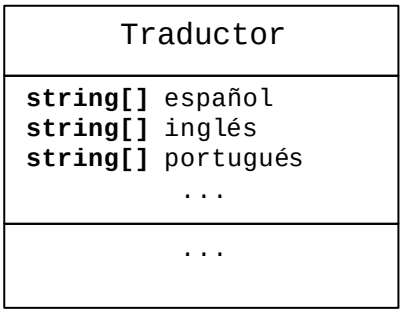
Ejemplo: Definir una clase que permita "traducir" palabras de Español a Inglés y Portugués (idea original en [24])

El elemento principal en la solución de este ejemplo es la determinación de la organización en memoria del "diccionario" que permitirá realizar la traducción. Aún cuando existen *Estructuras de Datos* (Parte III) o tipos de dato específicos para la representación de este tipo de información, en esta solución se aborda una representación muy simple para ejemplificar una aplicación interesante de los arreglos e indexers.

En la siguiente figura se muestra un ejemplo de la representación en memoria de la propuesta de diccionario que se desea implementar:



Del análisis de la figura se concluye que las palabras podrían ser representadas en arreglos por idiomas, donde en la i-ésima posición de cada arreglo se encuentra una palabra en cada uno de los respectivos idiomas. En un primer refinamiento la clase `Traductor` quedaría como se muestra en la siguiente figura.



Sugerimos definir las instancias de esta clase con un máximo de palabras predeterminado a través del constructor.

A partir de una instancia `traducir` de `Traductor`, En términos de funcionalidades sugerimos insertar las palabras de la siguiente forma:

```
Traducir["palabra"] = "word"; o Traducir["palabra", Inglés] = "word";
```

En el caso de las palabras en portugués se tiene una única opción:

```
Traducir[“palabra“,Portugués] = “palavra“;
```

Esto quiere decir que inglés es la traducción por defecto por lo que puede o no especificarse.

Análogamente la traducción de las palabras podría realizarse de la siguiente forma:

```
Console.WriteLine(Traducir[“palabra“]);

Console.WriteLine(Traducir[“palabra“, Inglés);

Console.WriteLine(Traducir[“palabra“, Portugués);
```

El análisis de las especificaciones anteriores conlleva a la necesidad de utilizar enumeraciones (para los idiomas) e indexers como se muestra en la variante de implementación que se muestra a continuación:

```
public enum Idioma
{
    Inglés,
    Portugués
}

public class Traductor
{
    string[] español;
    string[] inglés;
    string[] portugués;

    int cantPalabras;

    public Traductor(uint maxPalabras)
    {
          español = new string[maxPalabras];
          inglés = new string[maxPalabras];
          portugués = new string[maxPalabras];
    }

    public bool DiccionarioLLeno() {...}

    public string this[string palabra]
    {
          get
          {
                return this[palabra, Idioma.Inglés];
          }
          set
          {
                this[palabra, Idioma.Inglés] = value;
          }
    }

    public string this[string palabra, Idioma idioma]
    {
          get
          {
                ...
          }
          set
          {
```

```
            ...
        }
}
```

Corresponde ahora la implementación de las secciones **get** y **set** definidas con anterioridad. En el caso de **set** se necesita determinar la ubicación de la palabra en español (algoritmo de búsqueda, método posición) y posteriormente con el índice obtenido se puede retornar la traducción. Note que existen casos excepcionales cuando no existe la palabra a traducir o no existe traducción hasta entonces en el diccionario. El caso de la sección **set** aumenta la complejidad porque de forma similar a **get** primero hay que determinar si existe la palabra en español y luego si ya existe entonces la traducción. Seguidamente se muestran variantes de implementaciones de las secciones **get** y **set** definidas con anterioridad.

```
get
{
    string tempStr = null;
    int i = Posición(palabra);
    if (i != -1)
          switch (idioma)
          {
                case Idioma.Inglés: tempStr = inglés[i]; break;
                default: tempStr = portugués[i]; break;
          }
    else
          throw new Exception("Palabra no encontrada");

    if (tempStr == null)
          throw new Exception("No existe traducción");
    else
          return tempStr;
}
set
{
    int i = Posición(palabra);
    if (i == -1)
    {
          if (DiccionarioLLeno())
                throw new Exception("Diccionario lleno ...");

          i = cantPalabras++;
                español[i] = palabra.ToUpper();
    }

    switch (idioma)
    {
          case Idioma.Inglés:
                if (inglés[i] != null)
                      throw new Exception("Ya existe traducción al
                                                inglés de " + palabra);
                else
                      inglés[i] = value.ToUpper();
                break;
```

```
            default:
                  if (portugués[i] != null)
                        throw new Exception("Ya existe traducción al
                                                portugués " + palabra);
                  else
                        portugués[i] = value.ToUpper();
                  break;
      }
  }
```

Para concluir con la codificación de la clase `Traductor` faltaría la implementación de los métodos `Posición` y `DiccionarioLleno` que se muestran a continuación:

```
protected int Posición(string palabra)
{
    bool encontré = false;
    int i = -1;
    while (i + 1 < cantPalabras && !encontré)
          encontré = español[++i] == palabra.ToUpper();

    if (encontré)
          return i;
    else
          return -1;
}

public bool DiccionarioLLeno()
{
    return cantPalabras == español.Length;
}
```

Ejemplo: Definir una instancia de la clase `Traductor` que permita almacenar solamente 5 palabras y obtenga una aplicación que muestre los siguientes mensajes.

```
Insertando palabra y sus traducciones
Insertando casa y sus traducciones
Insertando zapatos y sus traducciones
Insertando leer y sus traducciones
Insertando nombre y sus traducciones
Insertando gato y sus traducciones
Diccionario lleno ...

Ya existe traducción al inglés de casa

Traduciendo palabra
palabra WORD PALAVRA
Traduciendo casa
casa HOUSE CASA
Traduciendo zapatos
zapatos SHOES SAPATOS
Traduciendo leer
leer READ LER
Traduciendo nombre
nombre NAME NOME
Traduciendo gato
Palabra no encontrada
```

```
{
    string[] español =
```

```
            {"palabra", "casa", "zapatos", "leer", "nombre", "gato"};
      string[] inglés =
            {"word", "house", "shoes", "read", "name", "cat"};
      string[] portugués =
            {"palavra", "casa", "sapatos", "ler", "nome", "gato"};

      Traductor traduce = new Traductor(5);

      try
      {
            for (uint i = 0; i < español.Length; i++)
            {
                  Console.WriteLine("Insertando {0} y sus traducciones",
                                          español[i]);
                  traduce[español[i]] = inglés[i];
                  traduce[español[i], Idioma.Portugués] = portugués[i];
            }
      }
      catch (Exception e)
      {
            Console.WriteLine(e.Message);
      }
      Console.WriteLine();

      try
      {
            traduce["casa"] = "house";
      }
      catch (Exception e)
      {
            Console.WriteLine(e.Message);
      }
      Console.WriteLine();

      try
      {
            for (uint i = 0; i < español.Length; i++)
            {
                  Console.WriteLine("Traduciendo {0}", español[i]);
                  Console.WriteLine(español[i] + " " +
                              traduce[español[i]] + " " +
                              traduce[español[i], Idioma.Portugués]);
            }
      }
      catch (Exception e)
      {
            Console.WriteLine(e.Message);
      }
      Console.ReadLine();
}
```

III.1.8 Ejercicios

2. Defina e implemente una clase `OperacionesCadenas` donde se implemente el algoritmo diseñado en el caso de estudio que se desarrolla en III.1.5. En lo adelante, todos los algoritmos que se diseñen para el trabajo con cadenas de caracteres (**`string`**) se implementaran en métodos de la clase `OperacionesCadenas`.

3. Dada una cadena de caracteres (cadena) y un carácter, determinar el número de ocurrencias de este en la referida cadena.

4. Dada una cadena, y un carácter (`c`), determinar el número de caracteres anteriores a la primera aparición de `c`.

5. Obtener la inversa de una cadena.

6. Dada una cadena que representa una frase, determinar la cantidad de palabras que la constituyen, dos palabras se separan por uno o más espacios y la frase puede comenzar o terminar con espacios.

7. Dada una cadena, una palabra y un entero `n` (`0 <= n < cadena.Length`) insertar la palabra en la posición `n` de la cadena.

8. Diseñe e implemente un algoritmo que dada una secuencia de caracteres cree otra secuencia siguiendo las siguientes normas: (idea original en [9])

- Después de un punto el carácter siguiente debe estar en mayúscula.
- Si hay dos blancos seguidos debe aparecer sólo uno.
- Después de una coma debe existir un blanco

9. Diseñe e implemente un algoritmo que determine si una cadena texto contiene las letras “A”, “B”, y “C” en este orden. Estas letras pueden estar separadas por otras. (idea original [9])

10. Generalice el algoritmo anterior para cualquier secuencia de letras.

11. Defina e implemente una clase que permita representar números a partir de una secuencia de ceros y unos de cualquier longitud y permita obtener su correspondiente valor entero en base 10.

12. Defina e implemente una clase `SecuenciaEnteros` que permita representar secuencias de valores enteros. En lo adelante, todos los algoritmos que se diseñen para el trabajo con secuencias de valores enteros se implementarán en métodos de la clase `OperacionesCadenas`.

13. Dada una secuencia de enteros, llamamos “meseta” a una subsecuencia de valores iguales de longitud ≥ 1. Diseñe e implemente un algoritmo que obtenga la longitud de la meseta de mayor longitud (idea original en [9]).

Por ejemplo, sea la siguiente secuencia `S`:

```
S = {35, 18, 18, 18, 5, 62, 35, 35, 11, 11, 11, 11, 9, 9}
```

La longitud de la meseta de mayor longitud es 4

14. Dada una secuencia de enteros, diseñe e implemente un algoritmo que genere una nueva secuencia formada por la secuencia de sumas parciales.

15. Dada una secuencia `S` de números enteros diseñe un algoritmo que determine si es creciente ($S_i \leq S_{i+1}$).

16. Diseñe una clase que permita representar conjuntos de enteros representados como secuencias e implemente las respectivas operaciones unión, intersección y diferencia.

17. Defina una clase que simule los resultados de un partido de tenis a partir de una secuencia de caracteres ‘1’ y ‘2’ que indica que jugador ha ganado cada punto. Permita obtener una salida que indica el marcador del partido después de cada punto jugado. Se supondrá que el partido lo gana el jugador que consigue seis juegos (idea original [9]).

Ejemplo:

Entrada: ’112211221’

Salida:
```
quince-nada
treinta-nada
treinta-quince
iguales a treinta
cuarenta-treinta
```

juego para el jugador 1
nada-quince
nada-treinta
quince-treinta

18. Dada una secuencia de enteros, escribe un algoritmo que indique si alguno de sus elementos coincide con la suma de los restantes elementos de la secuencia (idea original [9]).

. Ejemplo:

S = -1 2 4 -2 -1 8 3 –2 –3

Solución: 4

19. Refine la clase `Traductor` de forma tal que también se puedan insertan y traducir las palabras en cualquier "orden" de idioma.

Ejemplo:

```
traducir["palabra", Español, Inglés] = "word";
traducir["word", Inglés, Español] = "palabra";
Console.WriteLine(Traducir["palabra", Español, Inglés);
Console.WriteLine(Traducir["palavra", Portugués, Inlglés);
```

III.1.9 Bibliografía complementaria

- Capítulos 3, 4 y 5 de Jesús García Molina et al., Introducción a la Programación, Diego Marín ,1999.
- Capítulo 16 de Jose A. González Seco, El lenguaje de programación C#, 2002. Puede descargarse en http://www.josanguapo.com/

Capítulo III.2

III.2 Arreglos bidimensionales

Los arreglos descritos en el capítulo anterior, en los que se puede acceder a sus elementos utilizando un solo índice se denominan arreglos unidimensionales o de una sola dimensión. De la misma forma, si se tiene más de un índice, se denominan arreglos multidimensionales. En el caso particular de arreglos con dos índices se denominan arreglos bidimensionales o matrices y estos serán el objeto de análisis del presente capitulo.

Situación de análisis

Para el desarrollo de este capítulo se propone otra modificación en el dominio del problema particularmente para la clase `Estudiante`. En este caso se propone de cada asignatura almacenar las notas de los tres exámenes parciales para luego obtener la nota final a partir de aproximar el promedio de las notas parciales.

Sin dudas que la implementación de esta clase se facilitaría mucho reutilizando algunas clases propuestas en capítulos anteriores pero por objetivos didácticos se propone la representación explícita de las notas parciales como parte de las instancias de la clase `Estudiante` como se muestra a continuación:

`Resultados parciales`

4	5	3	4	4
4	5	4	3	5
4	4	4	3	4

`Notas`

4	5	4	3	4

Evidentemente en el caso de las `notas parciales` (`Resultados parciales`) se está en presencia de un arreglo bidimensional.

III.2.1 Arreglos bidimensionales en C#

Del capitulo anterior se conoce que utilizando el operador **`new`** es posible instanciar un arreglo y por consiguiente especificar el número de elementos del arreglo en tiempo de ejecución. Sin embargo, el número de dimensiones con que constará es estático y tiene que estar definido previamente.

En lo adelante se denominará rango de una dimensión al número de elementos que pueden ser almacenados en esta dimensión. El rango de elementos de cada dimensión no es algo estático sino dinámico, de ahí que es posible alterarlo cuando convenga. El cambio no obstante, implica en realidad la creación de un nuevo arreglo, con los elementos indicados y la liberación del arreglo anterior, lo que implica la pérdida de la información que este contuviese.

Un arreglo multidimensional es un arreglo donde los elementos se encuentran organizados en varias dimensiones. Para su definición se usa una sintaxis similar a la usada para declarar arreglos unidimensionales pero separando las diferentes dimensiones por comas. En lo particular un arreglo bidimensional es un arreglo de dos dimensiones.

Tómese como ejemplo la siguiente tabla que representa una abstracción o simulación de cómo se pudiesen representar los arreglos bidimensionales, aunque realmente en memoria no se representan de esa forma.

1	2	3
4	5	6

A continuación se muestran algunas formas de declarar e inicializar arreglos bidimensionales, tomando como caso de estudio la tabla anterior, es decir, se muestran diferentes formas de crear la tabla anterior en C# (la referencia devuelta producto de la creación se almacena en una variable `tabla`):

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

tabla = **new int**[,]{{1,2,3}, {4,5,6}};

tabla = **new int**[2,3]{{1,2,3}, {4,5,6}};

tabla = **new int**[2,3];
tabla[0,0] = 1; tabla[0,1] = 2; tabla[0,2] = 3;
tabla[1,0] = 4; tabla[1,1] = 5; tabla[1,2] = 6;

De forma análoga a los arreglos unidimensionales, para declarar un arreglo bidimensional en C#, se colocan corchetes entre el tipo y el nombre de la variable, sin embargo, ahora estos corchetes se separan por comas como se muestra a continuación:

<identificador de tipo> [,] <identificador de variable>;

De forma general, para definir un arreglo de dimensión `n`, se colocaran "`n-1`" comas entre los corchetes.

Pregunta: ¿Cómo imprimir el contenido de la variable `tabla` en el formato que se muestra a continuación?

1	2	3
4	5	6

En este caso es necesario conocer cómo acceder al elemento `(i,j)`; para ello se utilizan índices de forma similar a los arreglos unidimensionales sólo que ahora se utiliza un índice por cada una de las dimensiones `(tabla[i,j])`. Teniendo estos elementos en cuenta, se puede obtener el listado siguiente para imprimir la tabla en el formato deseado:

```
for (int i = 0; i < 2; i++)
{
    for (int j = 0; j < 3; j++)
        Console.Write("{0} ", tabla[i, j]);

    Console.WriteLine();
}
```

Situación de análisis

En el caso anterior fue muy fácil imprimir la variable `tabla` porque se conocía previamente el número de elementos de cada dimensión. Sin embargo, no es lo más común conocer el número de elementos de cada una de las dimensiones.

Pregunta: ¿Cómo imprimir el contenido de una variable `tabla` si no se conocen los elementos de cada una de las dimensiones?

Para ello se utiliza el método `GetLength` que retorna el número de elementos de cada dimensión como se muestra en el listado siguiente, esta solución produce los mismos resultados (para la misma variable `tabla`) que el código presentado con anterioridad:

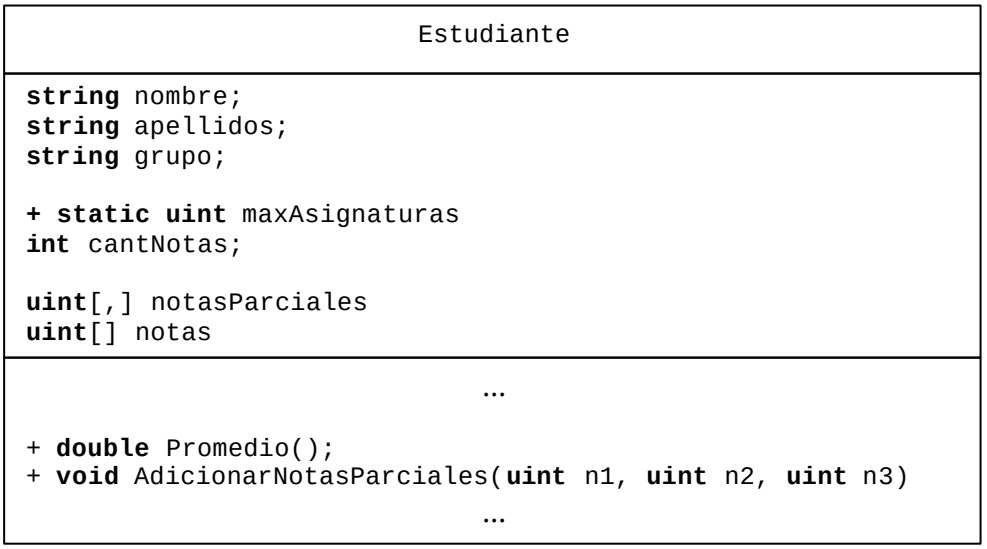
```
for (int i = 0; i < tabla.GetLength(0); i++)
{
    for (int j = 0; j < tabla.GetLength(1); j++)
          Console.Write("{0} ", tabla[i, j]);

    Console.WriteLine();
}
```

III.2.1.1 Caso de estudio: clase `Estudiante`, campo `notasParciales`

Ejemplo: Diseñar e implementar una clase `Estudiante` que represente la situación de análisis planteada en la introducción de II.2.

Para el diseño de esta clase se añadirán algunas restricciones al dominio del problema que se presenta en la referida situación de análisis. Por ejemplo, existe un máximo de asignaturas igual para todos los estudiantes (variable de clase `maxAsignaturas`) y por tanto un máximo de notas a obtener a partir de las ternas de notas parciales. Por tanto desde la creación de las instancias serán creados los arreglos `notas` y `notasParciales`, por lo que se debe tener un contador que registre el número de notas que se han actualizado (`cantNotas`) y este nunca puede ser mayor que `maxAsignaturas`.

En la siguiente figura se muestra el diseño de la referida clase.



En esta primera variante los arreglos `notasParciales` y `notas` se van llenando en la medida que se invoca al método `AdicionarNotasParciales` y se van introduciendo ternas que representas las notas parciales de las respectivas asignaturas. Seguidamente se muestra una variante de implementación de la clase `Estudiante`:

```
public class Estudiante
{
    string nombre;
    string apellidos;
    string grupo;

    static uint maxAsignaturas = 10;
    uint[,] notasParciales = new uint[3,maxAsignaturas];

    int cantNotas;
    uint[] notas = new uint[maxAsignaturas];

    public Estudiante(string nombre, string apellidos, string grupo)
    {
          this.nombre = nombre;
          this.apellidos = apellidos;
          this.grupo = grupo;
    }

    public void AdicionarNotasParciales(uint n1, uint n2, uint n3)
```

```
    {
        if (cantNotas == maxAsignaturas)
            throw new Exception("cantNotas == maxAsignaturas");

        notasParciales[0, cantNotas] = n1;
        notasParciales[1, cantNotas] = n2;
        notasParciales[2, cantNotas] = n3;

        notas[cantNotas++] =
            (uint)Math.Round((n1 + n2 + n3) / 3.0, 0);
    }

    public double Promedio()
    {
        double suma = 0;
        for (uint i = 0; i < cantNotas; i++)
            suma += notas[i];

        return (double)suma / cantNotas;
    }
}
```

Ejemplo: Obtener una aplicación consola que muestre los siguientes mensajes:

```
Resultados Parciales
4 5 3 4 4
4 5 4 3 5
4 4 4 3 4

 Notas
4 5 4 3 4

Promedio = 4
```

Donde la matriz se corresponde con las 3 notas parciales de un estudiante en 5 asignaturas y la fila independiente con las notas finales resultado del promedio de las parciales (ambos coinciden con los valores de la situación de análisis de la introducción de II.2) y por ultimo el promedio del estudiante.

Para que quede más claro el código de la aplicación se incluye un método (`NotasString`) en `Estudiante` que se ocupa de construir la matriz de las notas parciales y las notas finales, incluyendo los respectivos mensajes como se muestra seguidamente:

```
public string NotasString()
{
    string tempString = "Resultados Parciales \n";
    for (uint i = 0; i < notasParciales.GetLength(0); i++)
    {
        for (uint j = 0; j < cantNotas; j++)
            tempString += notasParciales[i,j] + " ";
        tempString += "\n";
    }

    tempString += "\n Notas \n";
    for (uint i = 0; i < cantNotas; i++)
        tempString += notas[i] + " ";

    return tempString;
}
```

Con el auxilio de este método la aplicación se reduce a crear la instancia de `Estudiante` y luego insertar las respectivas ternas de notas parciales; finalmente se imprimen los mensajes `NotasString()` y `Promedio()` como se muestra a continuación:

```
try
{
    Estudiante e = new Estudiante(...);

    e.AdicionarNotasParciales(4, 4, 4);
    e.AdicionarNotasParciales(5, 5, 4);
    e.AdicionarNotasParciales(3, 4, 4);
    e.AdicionarNotasParciales(4, 3, 3);
    e.AdicionarNotasParciales(4, 5, 4);

    Console.WriteLine(e.NotasString());
    Console.WriteLine();

    Console.WriteLine("Promedio = " + e.Promedio());

}
catch (Exception exc)
{
    Console.WriteLine(exc.Message);
}
```

Situación de análisis:

Es cuestionable el uso de un arreglo bidimensional en este caso, dado que se conoce previamente el número de exámenes parciales, luego con 3 arreglos unidimensionales se podría obtener una solución equivalente. No obstante, se ha fijado en 3 las notas parciales para simplificar esta primera implementación de la clase `Estudiante`. Sin embargo, este número no tiene por qué ser fijo.

Ejemplo: Refinar la clase `Estudiante` de forma tal que el número de notas parciales sea variable a nivel de clase pero coincida para todos los estudiantes y asignaturas:

A partir del enunciado anterior aparece una variable de clase que almacena el número de pruebas parciales que se haya definido para todos los estudiantes y asignaturas y por ello entonces el método `AdicionarNotasParciales` necesita como parámetro de un arreglo que tiene que coincidir en número de elementos con el número de pruebas parciales previamente fijado. A continuación se muestra una variante de implementación del refinamiento propuesto para la clase `Estudiante`.

```
public class Estudiante
{
    ...

    public static uint numPruebasParciales;
    public static uint maxAsignaturas = 10;
    uint[,] notasParciales =
                new uint[numPruebasParciales, maxAsignaturas];

    int cantNotas;
    uint[] notas = new uint[maxAsignaturas];

    public Estudiante(string nombre, string apellidos, string grupo)
    {...}

    public void AdicionarNotasParciales(uint[] notasParciales)
```

```
    {
        if (cantNotas == maxAsignaturas)
            throw new Exception("cantNotas == maxAsignaturas");

        if (notasParciales.Length != numPruebasParciales)
            throw new Exception("No concide número de notas
                                        parciales");

        double suma = 0;
        for (uint i = 0; i < numPruebasParciales; i++)
        {
            this.notasParciales[i, cantNotas] = notasParciales[i];
            suma += notasParciales[i];
        }

        notas[cantNotas++] =
            (uint)Math.Round(suma / numPruebasParciales, 0);
    }
    ...
}
```

Ejemplo: Utilizando el refinamiento a la clase `Estudiante`, obtener una aplicación que muestre mensajes similares a los mostrados usando la variante anterior de la clase consola, es decir:

```
Resultados Parciales
4 5 3 4 4
4 5 4 3 5
4 4 4 3 4

 Notas
4 5 4 3 4

Promedio = 4
```

En este caso se debe inicializar la variable de clase `numPruebasParciales` con el valor 3 que es el número de pruebas parciales que se ha predefinido, luego se debe crear la instancia de `Estudiante` y se le van insertando las ternas de notas parciales previamente colocadas en un arreglo que es la exigencia del método `AdicionarNotasParciales` para finalmente invocar a los métodos `NotasString` y `Promedio`.

```
try
{
    Estudiante.numPruebasParciales = 3;

    Estudiante e = new Estudiante(...);

    uint[] notasParciales1 = {4, 4, 4};
    e.AdicionarNotasParciales(notasParciales1);
    uint[] notasParciales2 = {5, 5, 4};
    e.AdicionarNotasParciales(notasParciales2);
    uint[] notasParciales3 = {3, 4, 4};
    e.AdicionarNotasParciales(notasParciales3);
    uint[] notasParciales4 = {4, 3, 3};
    e.AdicionarNotasParciales(notasParciales4);
    uint[] notasParciales5 = {4, 5, 4};
    e.AdicionarNotasParciales(notasParciales5);
```

```
    Console.WriteLine(e.NotasString());
    Console.WriteLine();
    Console.WriteLine("Promedio = " + e.Promedio());
}
catch (Exception exc)
{
    Console.WriteLine(exc.Message);
}
```

Note como hasta el momento los arreglos bidimensionales que se han implementado son *regulares* o *rectangulares*, es decir, para todas las dimensiones se tiene la misma cantidad de elementos. Es decir, para todas las filas se tiene el mismo número de columnas y viceversa.

III.2.2 Arreglos irregulares

Situación de análisis:

Se desea modificar el dominio del problema de la clase `Estudiante` sin prefijar el número de pruebas parciales a realizar, es decir por cada asignatura un estudiante puede realizar un número particular de exámenes parciales. Gráficamente podría verse como se muestra a continuación:

Resultados parciales

4	5	3	4	4
4	4		3	3
4			3	

Notas

4	4	3	3	4

Note en este caso la irregularidad del arreglo. Vale destacar que esta representación es un simple convenio, otra forma de ver los datos anteriores seria la siguiente, que incluso como se vera mas adelante, refleja con mas fidelidad la implementación en C#, dada nuestra asociación natural de "filas primera dimensión y columnas segunda dimensión".

Resultados parciales

4	4	4	
5	4		
3			
4	3	3	
4	3		

Notas

4
4
3
3
4

Nota: A fin de cuentas lo que se ha hecho es girar la tabla original.

En C# una tabla irregular no es más que una tabla cuyos elementos son a su vez tablas y su sintaxis es la siguiente:

```
<identificador de tipo> [][] <identificador de variable>;
```

La tabla irregular representada con anterioridad para las notas parciales podría definirse en C# de las siguientes formas:

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

int[][] notasParciales =
                  new int[][]{new int[]{4, 4, 4},
                              new int[]{5, 4},
```

```
                                new int[]{3},
                                new int[]{4, 3, 3},
                                new int[]{4, 3}};
    int[][] notasParciales = {new int[]{4, 4, 4},
                              new int[]{5, 4},
                              new int[]{3},
                              new int[]{4, 3, 3},
                              new int[]{4, 3}};
    int[][] nP = {new int[3],
                  new int[2],
                  new int[1],
                  new int[3],
                  new int[2]};
   nP[0][0] = 4; nP[0][1] = 4; nP[0][2] = 4;
   nP[1][0] = 5; nP[1][1] = 4;
   nP[2][0] = 3;
   nP[3][0] = 4; nP[3][1] = 3; nP[3][2] = 3;
   nP[4][0] = 4; nP[4][1] = 3;
```

Pregunta: ¿Cómo imprimir el contenido de la variable `nP` por ejemplo, en el formato que se muestra a continuación?

```
4  4  4
5  4
3
4  3  3
4  3
```

En este caso es necesario conocer cómo acceder al elemento `(i,j)`; para ello se utilizan índices pero como estas variables son arreglos de arreglos entonces se accede al "primero" y luego al "segundo" `(nP[i][j])`. Teniendo estos elementos en cuenta, se puede obtener el listado siguiente para imprimir la tabla en el formato deseado:

```
for (int i=0; i<nP.Length; i++)
{
    for (int j=0; j<nP [i].Length; j++)
          Console.Write("{0} ", nP[i][j]);
    Console.WriteLine();
}
```

III.2.2.1 Caso de estudio: clase `Estudiante`, campo `notasParciales`

Ejemplo: Diseñar e implementar una clase `Estudiante` que represente la situación de análisis planteada en II.2.2.

En este caso se mantienen algunas restricciones, por ejemplo, existe un máximo de asignaturas igual para todos los estudiantes (variable de clase `maxAsignaturas`) y por tanto un máximo de notas a obtener a partir de las notas parciales. También desde la creación de las instancias serán creados los arreglos `notas` y `notasParciales`, por lo que es necesario tener un contador que registre el número de notas que se han actualizado (`cantNotas`) y este nunca puede ser mayor que `maxAsignaturas`. Entonces el método `AdicionarNotasParciales` se ocupa de crear los respectivos arreglos dinámicamente para las notas parciales a través del arreglo que se le pasa como parámetro.

A continuación se muestra una aproximación de una variante de la clase `Estudiante` en función del dominio del problema definido:

```
public class Estudiante
{
    ...

    public static uint maxAsignaturas = 10;
    uint[][] notasParciales = new uint[maxAsignaturas][];

    int cantNotas;
    uint[] notas = new uint[maxAsignaturas];

    public Estudiante(string nombre, string apellidos, string grupo)
    { ... }

    public void AdicionarNotasParciales(uint[] notasParciales)
    {
          if (cantNotas == maxAsignaturas)
                throw new Exception("cantNotas == maxAsignaturas");

          double suma = 0;
          this.notasParciales[cantNotas] =
                                  new uint[notasParciales.Length];

          notasParciales.CopyTo(this.notasParciales[cantNotas], 0);

          for (uint i = 0; i < notasParciales.Length; i++)
                suma += notasParciales[i];

          notas[cantNotas++] =
                (uint)Math.Round(suma / notasParciales.Length, 0);
    }
}
```

Ejemplo: Utilizando el refinamiento a la clase `Estudiante`, obtener una aplicación que muestre los mensajes siguientes:

```
Resultados Parciales
4 4 4
5 4
3
4 3 3
4 3

Notas
 4
 4
 3
 3
 4

Promedio = 3,6
```

Para obtener esta aplicación los primeras instrucciones deben permitir la creación de la instancia de `Estudiante` y luego ir creando los arreglos con las secuencias de notas parciales que se actualizan a través del método `AdicionarNotasParciales` como se muestra a continuación:

```
Estudiante e = new Estudiante("A", "B", "C");
```

```
uint[] notasParciales1 = {4, 4, 4};
e.AdicionarNotasParciales(notasParciales1);

uint[] notasParciales2 = {5, 4};
e.AdicionarNotasParciales(notasParciales2);

uint[] notasParciales3 = {3};
e.AdicionarNotasParciales(notasParciales3);

uint[] notasParciales4 = {4, 3, 3};
e.AdicionarNotasParciales(notasParciales4);

uint[] notasParciales5 = {4, 3};
e.AdicionarNotasParciales(notasParciales5);
```

También para que quede más claro el código de la aplicación se incluye un método (`NotasString`) en `Estudiante` que se ocupa de construir la matriz de las notas parciales y las notas finales, incluyendo los respectivos mensajes como se muestra seguidamente:

```
public string NotasString()
{
     string tempString = "Resultados Parciales \n";
     for (uint i = 0; i < cantNotas; i++)
     {
       for (uint j = 0; j < notasParciales[i].Length; j++)
             tempString += notasParciales[i][j] + " ";
       tempString += "\n";
     }

     tempString += "\nNotas \n ";
     for (uint i = 0; i < cantNotas; i++)
       tempString += notas[i] + "\n ";

     return tempString;
}
```

Concluye entonces la aplicación de la siguiente forma:

```
Console.WriteLine(e.NotasString());
Console.WriteLine("Promedio = " + e.Promedio());
```

III.2.3 Caso de estudio: clase `Matriz`

Ejemplo: Definir una clase que permita representar matrices y sus correspondientes operaciones suma, producto y producto por un escalar.

Sin dudas que este es un clásico ejemplo de uso de arreglos regulares cuya implementación se describe a continuación:

```
public class Matriz
{
    double[,] elementos;

    public Matriz(double[,] elem)
    {
          elementos = new double[elem.GetLength(0), elem.GetLength(1)];

          for (uint i = 0; i < elem.GetLength(0); i++)
                for (uint j = 0; j < elem.GetLength(1); j++)
                      elementos[i,j] = elem[i,j];
    }
```

```
    public Matriz Suma(Matriz m) {...}

    public Matriz Producto(Matriz m) {...}

    public Matriz Producto(double a) {...}

    public Matriz Transpuesta(){...}

    public static Matriz operator +(Matriz m1, Matriz m2)
    {
          return m1.Suma(m2);
    }

    public static Matriz operator *(Matriz m1, Matriz m2)
    {
          return m1.Producto(m2);
    }

    public static Matriz operator *(Matriz m, double a)
    {
          return m.Producto(a);
    }

    public static Matriz operator *(double a, Matriz m)
    {
          return m.Producto(a);
    }

    public static Matriz operator !(Matriz m)
    {
          return m.Transpuesta();
    }

    public double this[uint i, uint j]
    {
          get
          {
                return elementos[i,j];
          }
    }

    public uint Filas
    {
          get
          {
                return (uint)elementos.GetLength(0);
          }
    }

    public uint Columnas
    {
          get
          {
                return (uint)elementos.GetLength(1);
          }
    }
}
```

En el listado anterior no se ha mostrado la implementación de las diferentes operaciones con matrices para no sobrecargarlo, las mismas se muestran a continuación:

```
public Matriz Suma(Matriz m)
{
    if (Filas != m.Filas || Columnas != m.Columnas)
          throw new Exception("Dimensiones diferentes ...");

    double[,] tempElems = new double[Filas, Columnas];
    for (uint i = 0; i < Filas; i++)
          for (uint j = 0; j < Columnas; j++)
                tempElems[i,j] = this[i,j] + m[i,j];

    return new Matriz(tempElems);
}

public Matriz Producto(Matriz m)
{
    if (Columnas != m.Filas)
          throw new Exception("No compatible producto...");

    double[,] tempElems = new double[Filas, m.Columnas];
    for (uint i = 0; i < Filas; i++)
          for (uint j = 0; j < m.Columnas; j++)
                for (uint k = 0; k < Columnas; k++)
                      tempElems[i,j] += this[i,k] * m[k,j];

    return new Matriz(tempElems);
}

public Matriz Producto(double a)
{
    double[,] tempElems = new double[Filas, Columnas];
    for (uint i = 0; i < Filas; i++)
          for (uint j = 0; j < Columnas; j++)
                tempElems[i,j] = this[i,j] * a;

    return new Matriz(tempElems);
}

public Matriz Transpuesta()
{
    double[,] tempElems = new double[Columnas, Filas];
    for (uint i = 0; i < Filas; i++)
          for (uint j = 0; j < Columnas; j++)
                tempElems[j, i] = this[i,j];

    return new Matriz(tempElems);
}
```

Ejemplo: Obtener una aplicación consola que muestre los siguientes mensajes:

```
m1 =
1 2 3
4 5 6
7 8 9
```

```
m2 =
1 1 1
1 1 1
1 1 1

m1 + m2 =
2 3 4
5 6 7
8 9 10

m2 * m2 =
3 3 3
3 3 3
3 3 3

m2 * 5 =
5 5 5
5 5 5
5 5 5

Tanspuesta m1 =
1 4 7
2 5 8
3 6 9
```

Los primeros pasos a dar en el diseño de la aplicación están relacionados con la creación de las respectivas instancias de la clase `Matriz`, seguidamente se muestra un variante:

```
double [,] auxElems = {{1,2,3},
                       {4,5,6},
                       {7,8,9}};
Matriz m1 = new Matriz(auxElems) ;

auxElems = new double[3,3]{{1,1,1},
                           {1,1,1},
                           {1,1,1}};
Matriz m2 = new Matriz(auxElems) ;
```

Seguidamente, y antes de mostrar los resultados de las operaciones se deben mostrar las matrices `m1` y `m2`. De forma análoga a las secciones anteriores, para que quede más claro el código de la aplicación se incluye un método (`ToString`) en la clase `Matriz` que se ocupa de convertir en una cadena los valores del arreglo bidimensional `elementos` que representa los valores de la matriz y que se muestra seguidamente:

```
public override string ToString()
{
    string temp = "";
    for (uint i=0; i < Filas; i++)
    {
          for (uint j=0; j<Columnas; j++)
                temp += this[i,j]+" ";
          temp += "\n";
    }

    return temp;
}
```

Finalmente, a través de este método se imprimen las matrices `m1` y `m2` y los resultados de las diferentes operaciones solicitadas como se muestra a continuación:

```
Console.WriteLine("m1 =");
Console.WriteLine(m1.ToString());

Console.WriteLine("m2 =");
Console.WriteLine(m2.ToString());

Matriz m3 = m1 + m2; // m1.Suma(m2)
Console.WriteLine("m1 + m2 =\n" + m3.ToString());

m3 = m2 * m2; // m2.Producto(m2);
Console.WriteLine("m2 * m2 =\n" + m3.ToString());

m3 = m2 * 5; // m2.Producto(5);
Console.WriteLine("m2 * 5 =\n" + m3.ToString());

m3 = !m1; // m1.Transpuesta()
Console.WriteLine("Tanspuesta m1 =\n" + m3.ToString());
```

III.2.4 Caso de estudio: clase `DiccionarioSinónimos`

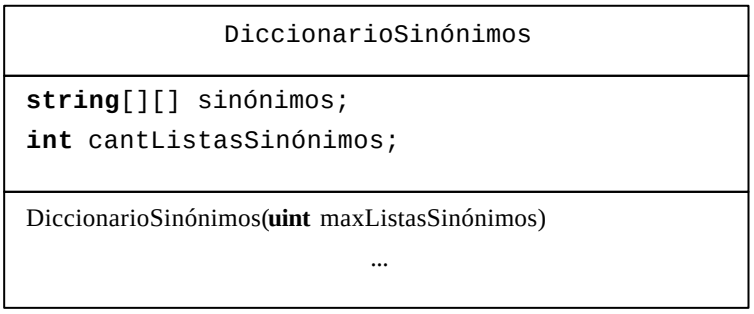
Ejemplo: Definir una clase que permita simular un Diccionario de Sinónimos.

Al igual que el caso de estudio del capitulo II.2 donde se desarrollo la clase `Traductor`, el elemento principal en la solución de este ejemplo es la determinación de la organización en memoria del "diccionario" que permitirá relacionar los sinónimos. Análogamente, aún cuando existen *Estructuras de Datos* (Parte III) o tipos de dato específicos para la representación de este tipo de información, en esta solución se aborda una representación muy simple para ejemplificar una aplicación interesante de los arreglos irregulares.

En la siguiente figura se muestra un ejemplo de la representación en memoria de la propuesta de diccionario que se desea implementar:

definir	precisar	concretar	puntualizar	
concluir	ultimar	sellar		
escuela	instituto	universidad	facultad	
estudiante	alumno	discípulo	escolar	educando
profesor	educador			

Del análisis de la figura o tabla anterior se concluye que las palabras podrían ser representadas en un arreglo irregular. Este arreglo podría ser interpretado como un arreglo de listas de sinónimos donde este último es a su vez otro arreglo. En un primer refinamiento la clase DiccionarioSinónimos quedaría como se muestra en la siguiente figura.



Vale destacar que en esta definición se limita la cantidad de listas de sinónimos para no desviar la atención del lector del objetivo principal. Esta limitación se concreta en el constructor de la clase, en la definición de un método DiccionarioLLeno, así como una variable cantListasSinónimos, que contiene la cantidad de listas de sinónimos almacenadas como se muestra a continuación:

```
public DiccionarioSinónimos(uint maxListasSinónimos)
{
      sinónimos = new string[maxListasSinónimos][];
}
public bool DiccionarioLLeno()
{
      return cantListasSinónimos == sinónimos.Length;
}
```

A partir de una instancia sinónimos de DiccionarioSinónimos, en términos de funcionalidades sugerimos insertar las listas de sinónimos de la siguiente forma (obviando las especificaciones sintácticas del lenguaje):

sinónimos[“definir “] = {precisar, concretar, puntualizar};

Análogamente obtención de los sinónimos podría realizarse de la siguiente forma:

Console.WriteLine(sinónimos[“definir “]);

El análisis de las especificaciones anteriores conlleva a la necesidad de utilizar indexers como se muestra en la variante de implementación que se muestra a continuación:

```
public class DiccionarioSinónimos
{
    string[][] sinónimos;
    int cantListasSinónimos;

    public DiccionarioSinónimos(uint maxListasSinónimos) {...}

    public bool DiccionarioLleno() {...}

    protected int Posición(string palabra) {...}

    public string[] this[string palabra]
    {
        get {...}

        set {...}
    }
    ...
}
```

Corresponde ahora la implementación de las secciones **`get`** y **`set`** definidas con anterioridad. En el caso de **`set`** se necesita determinar la ubicación de la `palabra` (algoritmo de búsqueda, método `Posición`) y posteriormente con el índice obtenido se puede retornar la correspondiente lista de sinónimos excluyendo `palabra`. Note que existen casos excepcionales cuando no existe la palabra a determinar sus sinónimos. El caso de la sección **`set`** se ha simplificado notablemente porque `palabra` y su correspondiente lista de sinónimos (**`value`**) se insertan en la primera posición vacía en el campo `sinónimos` sin verificar si ya existian o no y otros casos excepcionales. Seguidamente se muestran variantes de implementaciones de las secciones **`get`** y **`set`** definidas con anterioridad.

```
get
{
    string[] tempStrings = null;
    int i = Posición(palabra);
    if (i != -1)
    {
        tempStrings = new string[sinónimos[i].Length-1];
        for (uint j = 0, k = 0; j < sinónimos[i].Length; j++)
```

```
            if (sinónimos[i][j] != palabra.ToUpper())
                tempStrings[k++] = sinónimos[i][j];
        }
        else
            throw new Exception("Palabra no encontrada");

        return tempStrings;
    }
    set
    {
        if (!DiccionarioLleno())
        {
            sinónimos[cantListasSinónimos] = new string[value.Length+1];
            sinónimos[cantListasSinónimos][0] = palabra.ToUpper();
            for (uint i = 0; i < value.Length; i++)
                sinónimos[cantListasSinónimos][i+1]= value[i].ToUpper();
            cantListasSinónimos++;
        }
        else
            throw new Exception("Diccionario lleno");
    }
```

Seguidamente se presenta el método `Posición` utilizado en la sección **set** definida con anterioridad:

```
protected int Posición(string palabra)
{
    bool encontré = false;
    int i = 0;
    while (i < cantListasSinónimos && !encontré)
    {
        int j = 0;
        while (j < sinónimos[i].Length && !encontré)
              encontré = sinónimos[i][j++]==palabra.ToUpper();
        i++;
    }
    if (encontré)
        return --i;
    else
        return -1;
}
```

Ejemplo: Diseñar una aplicación que permita almacenar el diccionario de ejemplo que se presenta en el inicio de la sección:

definir	precisar	concretar	puntualizar	
concluir	ultimar	sellar		
escuela	instituto	universidad	facultad	
estudiante	alumno	discípulo	escolar	educando
profesor	educador			

En este caso se necesita crear una instancia sinónimos de `DiccionarioSinónimos` y luego insertar las diferentes listas de sinónimos como se presenta a continuación:

```
DiccionarioSinónimos sinónimos = new DiccionarioSinónimos(5);

string[] listaSinónimos = {"precisar", "concretar", "puntualizar"};
sinónimos["definir"] = listaSinónimos;

listaSinónimos = new string[]{"ultimar", "sellar"};
```

```
sinónimos["concluir"] = listaSinónimos;

listaSinónimos =
   new string[]{"instituto", "universidad", "facultad"};
sinónimos["escuela"] = listaSinónimos;

listaSinónimos =
    new string[]{"alumno", "discípulo", "escolar", "educando"};
sinónimos["estudiante"] = listaSinónimos;

listaSinónimos = new string[]{"educador"};
sinónimos["profesor"] = listaSinónimos;
```

Pregunta: ¿Cómo se podría obtener por pantalla el diccionario anterior aproximadamente de la siguiente forma?

```
definir precisar concretar puntualizar
concluir ultimar sellar
escuela instituto universidad facultad
estudiante alumno discípulo escolar educando
profesor educador
```

Realizando la siguiente operación:

```
Console.WriteLine(sinónimos.ToString());
```

Previa definición del método `ToString()` de la clase `DiccionarioSinónimos` como se muestra a continuación:

```
public string ToString()
{
      string tempString = null;

      for (uint i = 0; i < cantListasSinónimos; i++)
      {
            foreach (string s in sinónimos[i])
                  tempString += s + " ";
            tempString += "\n";
      }
      return tempString;
}
```

Pregunta: ¿Cómo se podrían obtener los sinónimos de “definir” de la siguiente forma?

```
Sinónimos de definir: precisar concretar puntualizar
```

El arreglo con los sinónimos de “definir” se puede obtener de la siguiente forma:

```
sinónimos["definir"]
```

Una vez obtenida esta secuencia se necesita transformar en una cadena para mostrar con `Console.WriteLine`. Para ello se define un método de clase que se utiliza de la siguiente forma:

```
string sinónimosStr =
   DiccionarioSinónimos.ListaSinónimosString(sinónimos["definir"]);
Console.WriteLine("Sinónimos de definir: {0}", sinónimosStr);
```

Y se define:

```
public static string ListaSinónimosString(string[] listaSinónimos)
{
      string tempString = null;
      for (uint i = 0; i < listaSinónimos.Length - 1; i++)
```

```
                tempString += listaSinónimos[i] + " ";

        return
            tempString + listaSinónimos[listaSinónimos.Length - 1];
    }
```

III.2.5 Ejercicios

1. Defina e implemente una clase `OperacionesCadenas` donde se implemente el algoritmo diseñado en el caso de estudio que se desarrolla en III.1.5. En lo adelante, todos los algoritmos que se diseñen para el trabajo con cadenas de caracteres (**`string`**) se implementaran en métodos de la clase `OperacionesCadenas`.

2. Defina la clase Traductor del capitulo III.1 utilizando arreglos bidimensionales. La representación del diccionario en este caso quedaría de la siguiente forma:

palabra	word	palabra
casa	house	casa
zapatos	shoes	sapatos
leer	read	ler
nombre	name	nome
...	...	...

3. Defina e implemente una clase `MatrizCuadrada`, que al menos tenga las siguientes funcionalidades:

- Sumatoria de los elementos de la diagonal principal.
- Sumatoria de los elementos de la diagonal secundaria.
- Sumatoria de los elementos de la triangular superior.
- Sumatoria de los elementos de la triangular inferior.
- Determine si es la matriz identidad.
- Determine si es la matriz nula.

4. En un torneo de ajedrez se conoce la cantidad de participantes, así como de cada jugador el nombre, fecha de nacimiento y un número que expresa su coeficiente ELO en el momento de desarrollar el torneo.

Dicho torneo se desarrolla de la forma todos contra todos a una vuelta y las estadísticas se controlan a través de una tabla de forma tal que el valor contenido en la casilla `(i,j)` representa el resultado de la partida desarrollada entre el jugador "`i`" y el jugador "`j`". En caso de victoria se le otorga un punto al ganador y cero al perdedor y en caso de empate cero a ambos jugadores.

Diseñe e implemente una clase que permita reflejar la situación anterior y que posibilite al menos:

- Obtener un listado con la ubicación de los competidores, defina usted mismo las estrategias de desempates en el caso de jugadores empatados en total de puntos.
- Determinar el participante más longevo y el de mayor coeficiente ELO (en ambos casos pudiera ser más de uno).

5. Se desea obtener el recorrido que debe hacer un caballo en un tablero de ajedrez de forma tal que recorra las 64 casillas sin pasar dos veces por ninguna.

Diseñe e implemente una clase que permita solucionar la problemática anterior. Utilice la regla de Warnedorff: "El caballo deberá moverse siempre hacia la casilla a partir de la cual existe una menos cantidad de continuaciones validas posibles".

6. Diseñe e implemente una clase que permita jugar *reversi*.

III.2.6 Bibliografía complementaria

- Capítulos 3, 4 y 5 de Jesús García Molina et al., Introducción a la Programación, Diego Marín ,1999.
- Capítulo 16 de Jose A. González Seco, El lenguaje de programación C#, 2002. Puede descargarse en http://www.josanguapo.com/

Tema IV

IV. RECURSIVIDAD. BUSQUEDA Y ORDENAMIENTO

Objetivos

- **Caracterizar** el esquema general de una iteración.
- **Definir** clases para tratar sucesiones.
- **Obtener** algoritmos iterativos y recursivos.
- **Implementar** algoritmos iterativos y recursivos.
- **Implementar** algoritmos simples de ordenamiento y búsqueda.
- **Obtener** aplicaciones que involucren instancias de las clases definidas en el desarrollo del tema.

Contenido

IV.1. Iteración y Recursividad.

IV.2. Búsqueda y Ordenamiento.

Descripción

El objetivo integrador de este tema es obtener aplicaciones en modo consola que involucren los contenidos presentados en el tema.

En el tema anterior se manejaron secuencias definidas explícitamente en el enunciado del problema, en la primera parte de este tema, se presenta el esquema general de una iteración, además se introducen los conceptos de iteración y recursividad.

En la segunda parte del tema se presentan algoritmos básicos de búsqueda y ordenamiento con el objetivo de continuar desarrollando el pensamiento algorítmico en los estudiantes.

Capítulo IV.1

IV.1 Iteración y Recursividad

"Cualquier iteración se puede interpretar como el recorrido de la secuencia de valores que toman las variables implicadas."[9]

Hasta el momento se ha aplicado la iteración para recorrer secuencias a las cuales se hace referencia explícita en los ejercicios. En este tema se aplicará la iteración a problemas cuyo enunciado no hace referencia explícita a la secuencia o secuencias que se necesitan recorrer.

Ejemplo: Dado dos números naturales se desea escribir un algoritmo que calcule su producto sin utilizar la operación de multiplicar.

Como se conoce el producto no es más que una suma abreviada, por lo tanto el producto de dos números naturales *a* y *b* se puede calcular sumando *a* veces *b* ó *b* veces *a*. Una solución al problema es recorrer y sumar cualquiera de las dos secuencias:

`a,a,a,a,a…..a(b veces)  ó b,b,b,b,b…..b(a veces)`

A continuación se muestra el código correspondiente a una de las variantes de solución.

```
public uint Multiplica(uint a , uint b)
{
    uint productoAB;
    for ( uint i = 1; i <= b; i++)
          productoAB += a;

    return productoAB;
}
```

En el código anterior se declaró una variable auxiliar *productoAB* que es la que almacenará la suma *b* veces de *a*, puede verse como no es necesario inicializarla a cero porque se inicializa por defecto.

Ejemplo: Calcular el máximo común divisor (*mcd*) entre dos números naturales *a* y *b* utilizando la propiedad:

$$mcd(a , b) = mcd(\ módulo(a-b) , b)$$

Antes de dar la solución del ejercicio vamos a ver la demostración de la propiedad que en él se plantea y al mismo tiempo ver como el conocimiento amplio de las matemáticas ayuda en la solución de algoritmos.

Demostración:

Sea $d = mcd(a , b)$, de donde "*d* divide *a* " (d/a) y "*d* divide *b* ".

d / a y $d / b \rightarrow d /$ *módulo* $(a – b)$**,** por lo tanto *d* divide al mismo tiempo a *b* y *módulo* $(a - b)$

Quedaría probar que *d* es el mayor de los divisores comunes de *b* y *módulo* $(a - b)$.

Supongamos que $d' > d$ es otro divisor común de *b* y *módulo* $(a - b)$.

d'/ b y $d'/$ *módulo* $(a – b) \rightarrow d'/ a$**,** lo cual quiere decir que *d'* es un divisor común de *a* **y** *b* que es mayor que *d* , esto es imposible porque *d* es el máximo común divisor de *a* y *b*. Luego no existe ningún divisor común de *b* y a *módulo* $(a - b)$ que sea mayor que *d.*

Veamos un ejemplo numérico:

$mcd\ (18 , 24) = mcd\ (18 , 6) = mcd\ (12 , 6) = mcd\ (6 , 6) = 6$

Este proceso se puede expresar a través del siguiente algoritmo:

```
Mientras a sea diferente de b
    Sustituimos el mayor por la diferencia
```

```
Al final en las dos variables queda el mismo valor que no es más
que el mcd(a , b)
```

El algoritmo se implementa a continuación.

```
public uint Mcd(uint a , uint b)
{
    while (a != b)
          if (a > b)
                a -= b;
          else
                b -= a;

    return a;
}
```

En el código anterior se ve como en el bucle "***while***" se controla que las instrucciones internas al mismo se ejecuten mientras las dos variables sean distintas. Las instrucciones internas al bucle lo que hacen es determinar cual de las dos variables es mayor y sustituirla por la diferencia con la otra variable. Al final se retorna cualquiera de las dos variables.

Ejemplo: Dado un número determinar si es entero.

Este ejercicio requiere de un poco de ingenio ya que no se dispone de una operación que se le pueda aplicar a un número y brinde directamente el resultado de si es entero ó no. Por tanto habrá que buscar alguna propiedad implícita que tengan los números enteros.

La clave para solucionar este ejercicio está en que desde cualquier número entero se puede llegar a otro número entero conocido sumándole o restándole repetidas veces 1. Tomemos como este entero conocido al número 0.

El algoritmo queda consta de los pasos siguientes:

```
Si el número es mayor que 0 entonces:
     Mientras sea mayor que 0
         Restarle 1
Sino
     Mientras sea menor que 0
          Sumarle 1
Al final si el resultado es 0 entonces es entero sino no lo es
```

A continuación se muestra el código correspondiente al algoritmo anterior.

```
public bool EsEntero(double n)
{
    if (n > 0)
          while (n > 0)
                n--;
    else while (n < 0)
                n++;

    if (n == 0)
          return true;
    else return false;
}
```

Puede verse como primero se determina si hay que sumar o restar 1 repetidas veces para llegar a 0 y después se chequea si se llegó o se pasó por él. Si se llega a 0 entonces el número es entero sino no lo es.

IV.1.1 Sucesiones

El término "sucesión" es muy difundido en el mundo de las matemáticas, en esta sección nos apoyaremos en algunas sucesiones de números para crear nuevas habilidades en la solución algoritmos.

Pregunta: ¿Qué se interpreta por sucesión de números?

Por sucesión de números se interpreta una secuencia aleatoria de números separados por coma (“,”).

Como ejemplo de sucesiones se tiene:

(I) 1 , 6 , -3 , 2.5 , 8 , 15

(II) 1 , 3 , 5 , 7 , 9 , ……

La primera tiene una cantidad finita de términos y no tiene una “ley de formación”, es decir no se puede deducir que término sigue a otro. La segunda tiene una cantidad infinita de términos y es fácil observar que representa la sucesión de los números impares.

Si en esta sucesión se hace:

$T_0 = 1$, se puede decir que la sucesión está dada por la ley de formación

$T_n = T_{n-1} + 2$,

por lo tanto cualquier término puede calcularse conociendo el anterior. De forma general las sucesiones en las que a partir de un término cualquiera todos los demás dependen de algunos o de todos los anteriores se denominan sucesiones recurrentes.

Una sucesión muy importante en el mundo de las matemáticas es la sucesión de Fibonacci:

1 , 1 , 2 , 3 , 5 , 8 , 13 , …….

donde cada término a partir del tercero es igual a la suma de los dos anteriores, formalmente:

$T_0 = 1$, $T_1 = 1$,

$T_n = T_{n-1} + T_{n-2}$, $n >= 2$

Pregunta: ¿Cómo determinar el n-ésimo término de esta sucesión?

Como puede observarse si n es igual a 1 o 2 el término es igual a 1 y sino se calcula sumando los dos términos anteriores. En general el término n-ésimo hay que irlo construyendo. Si se quiere calcular el quinto término hay que hacer:

$T_0 = 1$, $T_1 = 1$, $T_2 = 1 + 1 = 2$, $T_3 = 2 + 1 = 3$, $T_4 = 3 + 2 = 5$

Los pasos para determinar el n-ésimo término de esta sucesión son:

```
Inicializar dos variables a 1
aux1 = aux2 = 1
Repetir n-2 veces
   resultado = aux1 + aux2
   aux1 = aux2
   aux2 = resultado
```

La implementación del algoritmo se muestra a continuación:

```
public uint Finobacci(uint n)
{
    uint fibonacci = 1;
    uint aux1 = 1;
    uint aux2 = 1;
   for (uint i = 3; i <= n; i++)
   {
        fibonacci = aux1 + aux2;
        aux1 = aux2;
        aux2 = fibonacci;
   }
   return fibonacci;
}
```

En la implementación del método se ve como si n<3 no se entra en el ciclo for y se devuelve el valor que tiene por defecto la variable fibonacci que no es más que el valor 1. Una vez dentro del ciclo for lo que se hace es colocar en la variable fibonacci el resultado de la suma e ir moviendo las variables aux1 y aux2 por toda la sucesión.

Supongan que se va a calcular Fibonacci(5), en el paso de la iteración en que i vale 3 se hace fibonacci = 2, aux1 = 1 y aux2 = 2; cuando i es igual a 4 fibonacci = 3, aux1 = 2 y aux2 = 3, y cuando finalmente la i toma el valor 5 la variable fibonacci = 5, puede concluirse que las variables aux1 y aux2 son las que van recorriendo los valores que hay que sumar, mientras la variable fibonacci hace el papel de temporal y guarda el resultado final.

De forma general teniendo la ley de formación de cualquier sucesión recurrente es fácil calcular su n-ésimo término, solo hay que seguir el mismo esquema usado en el método Fibonacci pero con tantas variables como variables tenga la ley de formación.

IV.1.2 Recursividad

En esta sección se va a introducir una herramienta poderosa en la solución de problemas, la recursividad. Hasta el momento nuestro pensamiento para resolver los problemas es totalmente iterativo, es decir se reduce a ejecutar instrucciones de forma secuencial o alternativa, recorrer un arreglo, ejecutar un bucle mientras se cumpla una condición o una combinación de estos.

Para acercarnos un poco a la recursividad vamos a responder la siguiente pregunta.

Pregunta: ¿Qué algoritmo seguiría para subir una escalera?

Si rápidamente se piensa en implementar un método que suba una escalera suponiendo que existe un método `Subir` que sube un escalón se haría:

```
Desde i = 1 hasta la cantidad de escalones
    Subir
```

Esta solución al problema es iterativa pero no refleja del todo la realidad del proceso implícito que se sigue para subir una escalera, en efecto lo que se hace es:

```
Si queda un escalón entonces se sube y se termina
sino, se sube un escalón y se vuelve al principio
```

Volver al principio significa realizar el mismo análisis pero esta vez con un escalón menos.

Si se codifica este proceso suponiendo que existe un método `Subir`() sin parámetros que sube un escalón quedaría como sigue:

```
public void SubirEscalera(int cantEscalones)
{
    if (cantEscalones == 1)
          Subir();
    else
    {
          Subir();
          SubirEscalera(cantEscalones - 1);
    }
}
```

En el interior del código del método `SubirEscalera` se hace una llamada a el mismo. Cuando ocurre esto se dice que el método es *recursivo*.

Cuando se hace que un método sea recursivo se está buscando, generalmente, una forma simple de realizar una cierta operación que de otra forma sería más compleja [21].

Situación de análisis

Como ejemplo de recursividad analicemos una variante de solución para el método `Factorial` visto en secciones anteriores e implementado de forma iterativa.

El método `Factorial` cumple las siguientes propiedades:

(I) $F_0 = 1$,

(II) $F_n = n*F_{n-1}$, $n > 0$

La propiedad (II) es en sí la propiedad recursiva y la propiedad (1) es la condición de parada puesto que si no se invocaría al método `Factorial` "infinitamente" sin obtener resultado alguno.

Se ha colocado infinitamente entre comillas porque en realidad cada vez que se llama a un método, indistintamente del punto desde el que se realice la llamada, en la pila se almacena una dirección de retorno, así como el espacio necesario para las variables locales que tenga el método. De esta forma un método recursivo que se llame sin condición de parada provocará un desbordamiento de pila.

La implementación recursiva del método `Factorial` queda como sigue:

```
public int Factorial(int n)
{
    if (n == 0)
          return 1;
    else return n*Factorial(n-1);
}
```

Suponga que se invoca al método `Factorial` para calcular el factorial de 4, como 4 es diferente de cero el método retorna el resultado de multiplicar 4 por `Factorial(3)`, pero para conocer este valor primero hay que calcular el factorial de 3, que no es más que 3 por `Factorial(2)` y así sucesivamente hasta que se llegue a 0. El llamado a `Factorial(0)` ya deja de invocar al método `Factorial` y retorna el valor 1, este valor es sustituido en la expresión 1*`Factorial(0)` retornándose así `Factorial(1)` y sustituyéndose a su vez en 2*`Factorial(1)`, la secuencia de sustituciones para cuando se llega a 4*`Factorial(3)` que es el valor retornado por el la primera llamada al método `Factorial`.

Este proceso se ve a continuación:

```
Factorial(4)
return 4 * Factorial(3)
         return 3 * Factorial(2)
                  return 2 * Factorial(1)
                           return 1 * Factorial(0)
                                    return 1
```

Es importante destacar que además de que en la pila se reserve memoria para las variables locales de cada llamado, también se guarda la dirección de retorno del mismo, esto significa que cuando cada llamado termina la ejecución del programa sigue en la instrucción siguiente del llamado anterior.

Situación de análisis

Desarrollar una solución recursiva para la sucesión de Fibonacci

Como ya se vio la sucesión de Fibonacci cumple para *n* mayor o igual que 3 que cada término es igual a la suma de los dos anteriores.

En primer lugar hay que determinar la condición de parada del algoritmo. La sucesión define a sus dos primeros términos con el valor 1, por lo que la condición de parada tiene que ser "`(n==1 || n==2)`". Esto quiere decir que si el término es uno de los dos primeros el método devolverá el valor 1, en caso contrario retornará la suma de `Fibonacci(n-1) + Fibonacci(n-2).`

A continuación se muestra la implementación del método `Fibonacci.`

```
public uint Fibonacci(uint n)
{
    if (n == 1 || n == 2)
          return 1;
    else return Fibonacci(n-1) + Fibonacci(n-2);
}
```

A continuación se muestra el esquema de una llamada al método `Fibonacci` para el valor 4.

`Fibonacci(4)`

return `Fibonacci(3)` + `Fibonacci(2)`

return `Fibonacci(2)` + `Fibonacci(1)` **return** 1

return `1` **return** 1

En el esquema puede notarse como se desencadena el llamado hasta encontrar la condición de parada, en este caso "`(n==1 || n==2)`" y hasta que no está calculado `Fibonacci(3)` no se pasa al segundo operando, en este caso `Fibonacci(2).`

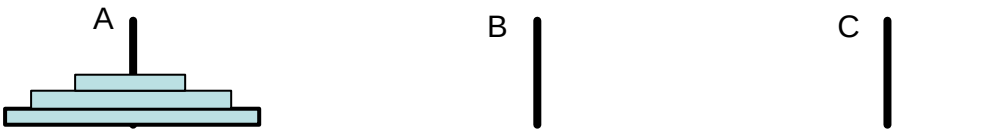
IV.1.3 Caso de estudio: `Torres de Hanoi`

El problema de las *Torres de Hanoi* es un ejemplo clásico que ilustra la potencia de la recursividad para la solución de problemas. El enunciado es el siguiente:

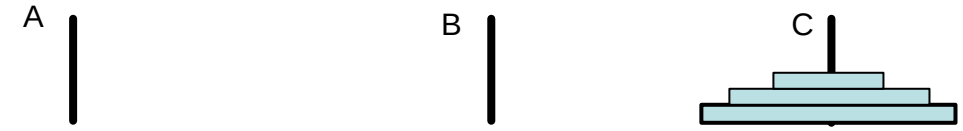
Se tienen tres clavijas (*torres*) A, B y C. En la clavija A se han colocado *n* discos de diámetros diferentes, de forma tal que cada disco de un mayor diámetro queda debajo de los que tienen menor diámetro que él (véase la figura). El objetivo final es mover los *n* discos de la torre A a la C siguiendo las reglas:

i. En cada caso se toma el disco que está en el tope de una de las torres y se pasa hacia otra torre.
ii. En ningún momento deberá haber un disco encima de otro de menor diámetro que él.

Diseñe e instrumente una solución recursiva al problema.



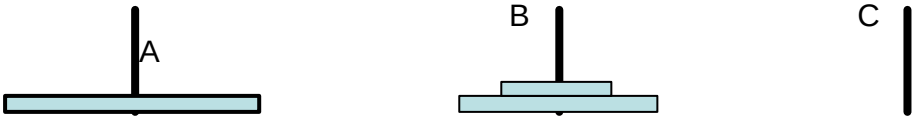
`Estado inicial`



`Estado final`

Como se enunció el objetivo del problema es mover los *n* discos de la torre A a la C. Suponiendo que se hallan pasado todos los discos de la torre Aa la C cumpliendo con las reglas, el disco que estaría en la base de la torre C sería el mismo que estaba en la base de la torre A.

Pregunta: ¿Cómo pasar el disco base de la torre A a la C si sobre él hay otros discos?



Para lograrlo se tendría que haber llegado a una configuración como la que se muestra a continuación,

es decir, haber movido *n-1* discos hacia la torre B, para luego mover el disco base de A a C y nuevamente mover los *n-1* discos de la torre B a la torre C.

A su vez el proceso de mover los *n-1* discos se descompone en los mismos tres pasos pero cambiando el origen y el destino. Los pasos a seguir son:

1. Mover *n-1* discos de la torre A(origen) a la B (destino), utilizando a C como auxiliar.
2. Mover disco base de la torre A(origen) a la C (destino).
3. Mover *n-1* discos de la torre B(origen) a la C (destino), utilizando a A como auxiliar.

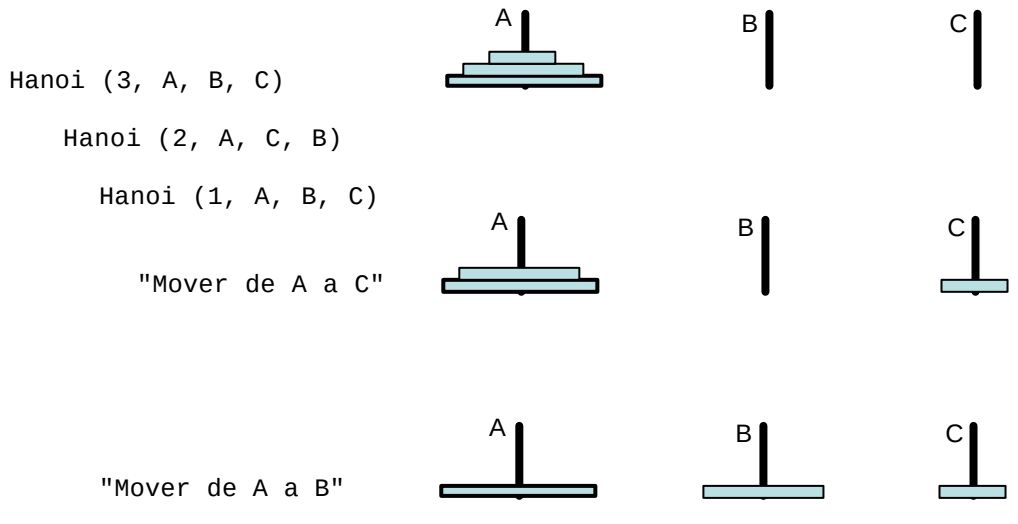
Como ya se evidencia, no todos estos pasos se pueden realizar en un mismo movimiento, pues en el primero y tercero hay que mover más de un disco. Este proceso nos lleva a un algoritmo recursivo que consta de los siguientes pasos:

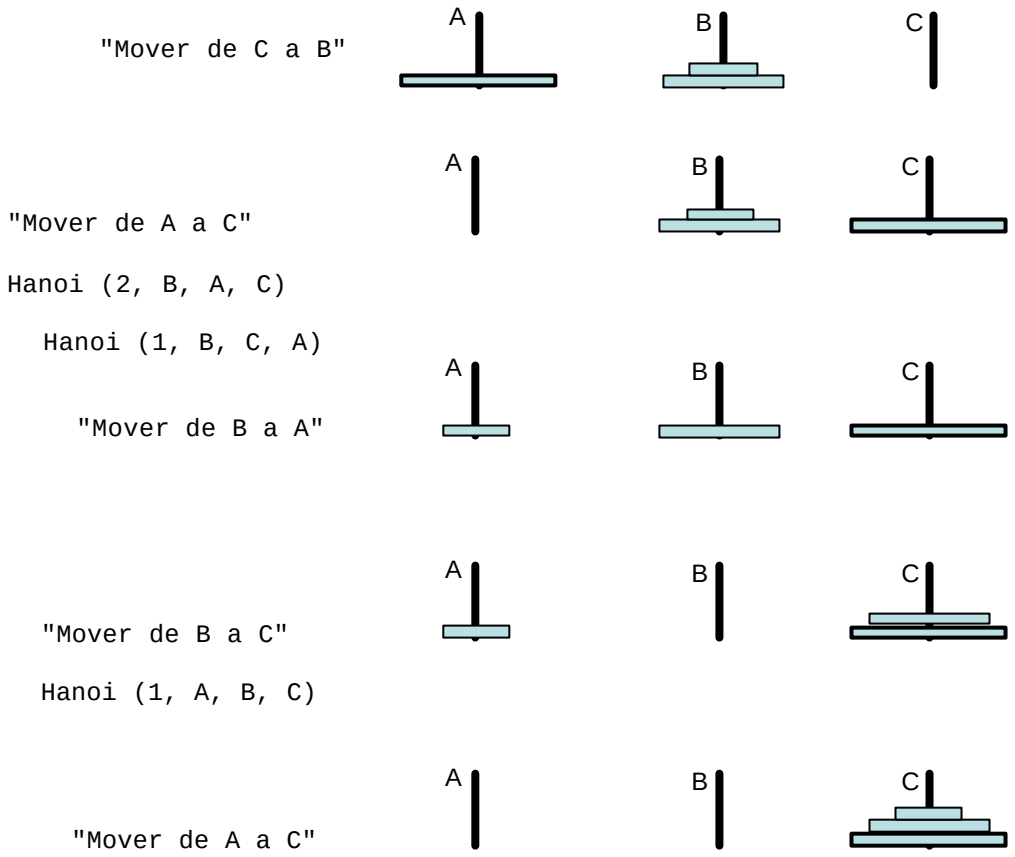
```
Si la cantidad de discos a mover es uno
     Mover el disco del origen al destino
Sino
     Mover n-1 discos del origen al auxiliar
     Mover un disco del origen a destino
     Mover n-1 discos del auxiliar al destino
```

Nota: Es importante ver que el origen, el destino y la torre auxiliar no siempre son los mismos.

A continuación se implementa el proceso anterior y se ejemplifica gráficamente para `n = 3`.

```
public void Hanoi(uint n,string origen,string auxiliar,string destino)
{
   if (n == 1)
      Console.WriteLine("Mover de {0} a {1}",origen,destino);
   else
   {
      Hanoi(n-1,origen,destino,auxiliar);
      Console.WriteLine("Mover de {0} a {1}",origen,destino);
      Hanoi(n-1,auxiliar,origen,destino);
   }
 }
```





Se propone al lector probar el algoritmo con otros valores, p ara que se relacione con la belleza y potencialidad de la recursividad.

IV.1.4 Ejercicios propuestos

1. Escriba un método que calcule el número de combinaciones en que pueden tomarse *m* objetos de *n* ,dada la formula :

   $C_{n,m} = n! / ((n – m)! * m!)$

2. Escriba un método recursivo con el mismo objetivo del ejercicio anterior que utilice la identidad :

   $C_{n,m} = C_{n - 1,m - 1} + C_{n - 1,m}$

3. Programe un método que dado un número x determine el valor de la función exponencial , usando la expresión:

   $e^x = x^0 / 0! + x^1 / 1! + x^2 / 2! + ……. + x^n / n!$

   la precisión absoluta (*n*) deseada se recibe como parámetro.

4. ¿Qué hace la siguiente función matemática?

$$Q(a, b) = \begin{cases} 0 & \text{si } a<b \\ Q(a-b, b)+1 & \text{e.o.c} \end{cases}$$

   Programe un método recursivo y uno iterativo que permitan evaluar la función para un a y b dados.

5. Escriba un método que reciba una cadena de caracteres y la invierta.

6. Escriba un método que, utilizando el método implementado en el ejercicio anterior, determine si una palabra dada es o no palíndromo.

7. El conjunto de *Wirth(W)*, se define a través de las siguientes reglas:

   I. 1 I W.

   II. si x I W, entonces 2*x + 1 I W y 3*x +1 también pertenece.

Escriba un método que determine e imprima los 100 primeros números naturales que son miembros del conjunto de *Wirth*. Pruebe a encontrar solución recursiva e iterativa.

IV.8 Bibliografía complementaria

- Capítulos 8 de Jesús García Molina et al., Introducción a la Programación, Diego Marín ,1999.

Capítulo IV.2

IV.2 Búsqueda y Ordenamiento

IV.2.1 Búsqueda

La búsqueda de un objeto específico en una lista de objetos es un procedimiento que se ha llevado a cabo a lo largo del texto. En todos los casos el algoritmo a seguir ha sido buscar uno por uno hasta encontrar el objeto deseado como se muestra a continuación.

```
public bool Buscar(int n)
{
   for (int i = 0; i < valores.Length; i++)
    if (valores[i] == n)
       return true;
   return false;
}
```

Nota: Si el objeto no está en el arreglo no se sabe hasta que no se recorra completo.

Pregunta: ¿Qué sucede si se tiene el arreglo de objetos ordenado ascendentemente?

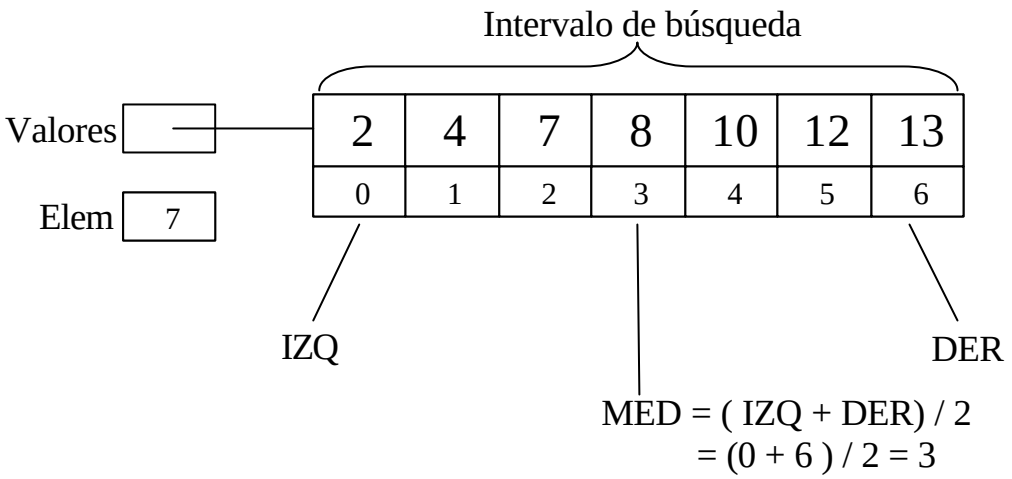
Si el arreglo está ordenado ascendentemente se puede saber sin recorrerlo si el objeto no está con solo chequear si es menor que el primer objeto ó mayor que el último. Además en caso de que el objeto pudiera estar en el arreglo solo habría que recorrerlo mientras el objeto buscado sea mayor que el de la posición actual. Si en algún momento llega a ser menor entonces se puede concluir que no está en el arreglo.

```
public bool BuscarOrdenado(int n)
{
   if (n < valores[0] || n > valores[valores.Length - 1])
      return false;
   else
   {
    int i = 0;
    while (i < valores.Length && n >= valores[0])
          if (valores[i++] == n)
             return true;
    return false;
   }
}
```

IV.2.1.1 Búsqueda binaria

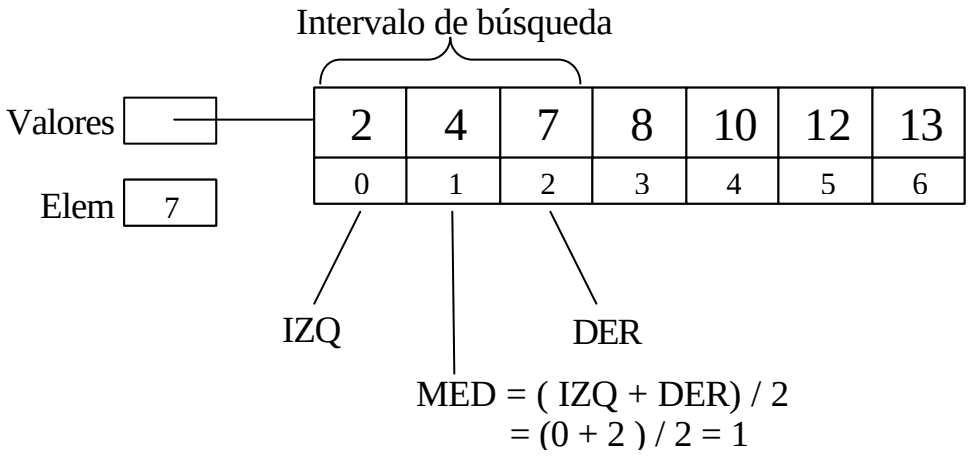
Un algoritmo de búsqueda muy interesante en arreglos ordenados es la *búsqueda binaria*, que consiste en tomar el objeto que ocupa en el arreglo la posición del medio (si la cantidad de objetos es impar se toma el que ocupa la posición mitad - 1), si el objeto que se busca es mayor entonces hay que buscarlo en la mitad de la derecha (queda claro que no puede estar en la otra mitad) y si es menor en la mitad de la izquierda. Este proceso se repite hasta encontrar el número o concluir que no está.

Si se busca el número 7 en la secuencia { 2 , 4 , 7 , 8 , 10 , 12 , 13 } los pasos serían :



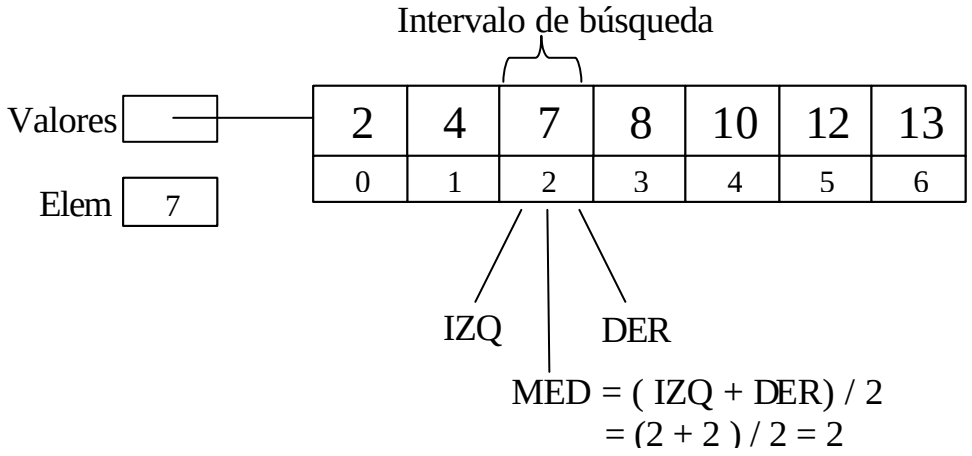
Tomar el 8 y compararlo con el 7

Como el 8 es mayor hay que buscarlo del 2 al 7



Tomar el 4 y compararlo con el 7

Como el 4 es menor hay que buscarlo del 7 al 7



Tomar el 7 y compararlo con el 7

Luego el 7 está en el arreglo

En este esquema de búsqueda se realiza siempre la misma acción que es buscar el número en un intervalo que varía en dependencia de si el número buscado sea mayor o menor que el que ocupa la posición del medio

El algoritmo cuenta con dos condiciones de parada , la primera que se encuentre el número y la segunda que el número no esté(esta condición se puede chequear analizando que el intervalo en el que hay que buscar tenga longitud positiva).

Todo este análisis conduce a una solución recursiva para el algoritmo, la misma se presenta a continuación.

```
public bool BusquedaBinaria(int n, uint indice1, uint indice2)
{
   int mitad = (indice1 + indice2)/2;
   if (n == valores[mitad])
      return true;
   if (indice1 > indice2)
      return false;
   if (n > valores[mitad])
      return BusquedaBinaria(n, mitad+1 , indice2);
   else return BusquedaBinaria(n, indice1, mitad-1);
}
```

En la implementación anterior se ve como si `(indice1 > indice2)` se retorna falso, es decir que el número no se encuentra en el arreglo, se sugiere al lector que pruebe el algoritmo para algunos juegos de datos para que compruebe que cuando el número no está en el arreglo los índices se cruzan.

IV.2.2 Ordenamiento

El proceso de ordenar una lista de objetos ascendente o descendentemente es fundamental y se realiza con mucha frecuencia en la solución de problemas.

Cuando se habla de ordenar lo primero que viene a la mente es ordenar números, pero en la vida diaria ordenamos distintos tipos de objetos siguiendo diversos criterios.

Por ejemplo las palabras *mesa*, *computadora* y *artista* se pueden ordenar por orden alfabético

artista, *computadora* y *mesa*

o por su longitud

mesa, *artista* y *computadora*

Como ya se vio en secciones anteriores C# permite la sobrecarga de operadores, luego es posible sobrecargar el operador <= (>= , < , > , == , !=) y de esta forma comparar dos objetos según el criterio deseado. A continuación se verán algunos esquemas simples de ordenamiento que funcionan independientemente del tipo de objeto que se esté ordenando, debido a esto, sin perder generalidad trabajaremos los ejemplos con arreglos de enteros.

IV.2.2.1 Ordenamiento por burbujas

El método más usado para ordenar listas de objetos no demasiado grandes es el método de burbujas. La idea básica que hay detrás del nombre es imaginar que el arreglo a ordenar se coloca verticalmente y que los objetos menos “pesados” suben más rápidamente hacia el tope del arreglo como las burbujas menos pesadas lo hacen a la superficie.

Se hacen repetidos pases sobre el arreglo desde el final hacia el principio. Si dos objetos adyacentes están desordenados entonces se intercambian de posición. El efecto de esta operación es que en el primer pase el objeto menos “pesado” es llevado hacia la primera posición; en el segundo pase, el segundo objeto menos pesado (que no es más que el menos “pesado” de los que quedan) es llevado a la segunda posición. De forma general en el pase i-ésimo el i-ésimo objeto menos “pesado” es llevado a la posició *i*.

A continuación se presenta el código que lleva a cabo el proceso descrito, en el mismo se ordena un arreglo de enteros.

```
public void Burbujas(int[] valores)
{
   for (int i = 0; i <= valores.Length - 2; i++)
        for (int j = valores.Length - 1; j >= i+1; j--)
        {
             if (valores[j] < valores[j-1])
                 Intercambia(ref valores[j],ref valores[j-1])
        }
}
```

A medida que la *i* va avanzando el arreglo va quedando ordenado hasta la posición *i -1* y a medida que la *j* va decreciendo se va llevando el menor valor de los que quedan hasta la posición *i*.

El método Intercambia es usado en este método y en la sección posterior, por lo que es presentado a continuación.

```
public void Intercambia(ref int x , ref int y)
{
    int aux;
     x = y;
     y = aux;
}
```

Si se aplica el método de burbujas para ordenar los números {8, 4, 5, 9, 7, 2} se puede ver en la figura como queda el arreglo después de cada pasada, la línea indica que los elementos que están por encima de ella ya están ordenados y en su posición.

A pesar de ser 6 elementos el número de pasadas es 5 porque después de ubicar los 5 menores en sus respectivas posiciones no queda otra opción para el mayor que no sea la última, la misma lógica del algoritmo va llevando el mayor elemento a la última posición del arreglo.

8	2	2	2	2	2
4	8	4	4	4	4
5	4	8	5	5	5
9	5	5	8	7	7
7	9	7	7	8	8
2	7	9	9	9	9
Inicio	i =1	i =2	i =3	i =4	i =5

IV.2.2.2 Ordenamiento por inserción

El segundo esquema de ordenación que vamos a tratar es el llamado método de inserción que tiene su fundamento en el siguiente planteamiento: Si se tiene una secuencia de *n* elementos a_1 , a_2 , ….. , a_n de forma tal que $a_1 <= a_2 <= ….. <= a_n$, es decir, ordenados de menor a mayor, e insertamos un elemento a_k en una posición tal que $a_i <= a_k <= a_{i+1}$ para algún *i* (1 <= *i* <= *n*) entonces la nueva secuencia a_1, a_2 , …, a_i , a_k , a_{i+1}....., a_n sigue estando ordenada.

Basado en esto el método comienza tomando el segundo elemento e insertándolo en su correspondiente posición en la secuencia formada por el primer elemento, esto se logra intercambiando el segundo elemento con el primero en caso de que estén desordenados; después toma el tercer elemento y lo inserta en la secuencia formada por los dos primeros elementos (la cual ya está ordenada) realizando tantos intercambios como sean necesarios. De forma general el método inserta el *i-ésimo* elemento en la secuencia formada por los *i-1* elementos anteriores, los cuales ya están ordenados.

A continuación se muestra el código que lleva a cabo el algoritmo descrito.

```
public void Insercion(int[] valores)
{
   for (int i = 1; i <= valores.Length - 1; i++)
   {
    int j = i ;
    while(j >= 1 && valores[j] < valores[j-1])
    {
        Intercambia(ref valores[j],ref valores[j-1]);
        j = j – 1;
    }
   }
}
```

El código describe claramente el pro ceso por lo que no es necesario un comentario adicional. Solo resaltar el chequeo de (*j >=1*) para evitar que la *j* se valla de rango en la condición (*valores[j] > valores[j-1]*).

A continuación se muestra el método de inserción aplicado a la secuencia de números de la sección anterior.

8	4	4	4	4	2
4	8	5	5	5	4
5	5	8	8	7	5
9	9	9	9	8	7
7	7	7	7	9	8
2	2	2	2	2	9
Inicio	i =2	i =3	i =4	i =5	i =6

Tema V

V. APENDICES

Objetivos

- **Caracterizar** los conceptos básicos de la plataforma de desarrollo Microsoft .NET.

Contenido

V.1. Plataforma de desarrollo Microsoft .NET.

Descripción

El objetivo de este tema es brindar un conjunto de elementos complementarios para la mejor comprensión del texto y el uso más eficiente del lenguaje y el entorno de trabajo utilizado.

De momento apenas se presentan los conceptos básicos de la plataforma .NET, pues este es el entorno que sustenta a C#. En el futuro inmediato se deben abordar, al menos, los elementos relacionados sobre cómo documentar los códigos y por ende introducir elementos de XML y atributos.

Capítulo V.1

V.1 Plataforma de desarrollo Microsoft NET

Microsoft diseñó C# para aprovechar el nuevo entorno o plataforma .NET (NET Framework o simplemente .NET). Como C# forma parte de este nuevo mundo .NET, es imprescindible entonces comprender lo que proporciona esta plataforma para el desarrollo de aplicaciones de software y de que manera aumenta su productividad [10].

Sin un conocimiento básico de las características esenciales de la plataforma .NET y del papel que juega C# en este iniciativa de Microsoft, será más difícil la comprensión de los elementos centrales de C# que supone se apoyen en el entorno de ejecución .NET [13].

Con anterioridad en el capítulo I.2 justificamos nuestra elección para utilizar C# como lenguaje para un primer curso de programación desde el enfoque OO. Teniendo en cuenta el diseño de este lenguaje con el objetivo de aprovechar al máximo las bondades de la tecnología .NET es imprescindible presentar una visión general sobre la referida tecnología pues constantemente estaremos haciendo uso de sus conceptos.

V.1 Evolución hacia .NET

La primera parte de esta sección se desarrolla respondiendo a tres ejercicios a partir de las ideas desarrolladas en [24].

Pregunta: ¿Cuál ha sido la evolución de los modelos de programación?

> Las primeras aplicaciones de software fueron diseñadas para Mainframe\minicomputadoras y seguidamente para Desktops PCs. Posteriormente y con el surgimiento y desarrollo de las redes de computadores se establece el paradigma de programación distribuida: en un primer momento en dos niveles (Cliente-Servidor) y luego en tres niveles, introduciéndose un nivel intermedio (CORBA, DCOM, RMI).

Pregunta: ¿Cuál ha sido la evolución de Internet?

> Internet fue creada en un principio para tratar contenido estático entre los clientes Web. Estas páginas Web nunca cambiaban y eran las mismas para todos los clientes que navegaban hasta esa localización. Es por ello que esta primera generación puede ser caracterizada por la presencia de páginas estáticas con contenido pasivo, aparecen entonces los primeros estándares (HTML) y navegadores (Netscape).
>
> Una segunda generación en el uso de Internet estuvo caracterizada por el boom de Java, la aparición de los lenguajes de *script* y la conexión a Bases de Datos. En este momento las páginas comienzan a ser dinámicas y aparecen tecnologías como Microsoft ASP y Java JSP, aparecen además las primeras herramientas para desarrollar páginas Web.
>
> En lo particular en esta segunda generación Microsoft lanzó servidores activos para permitir la creación de páginas dinámicas basadas en la aportación e interacción del usuario con la página Web [10].
>
> La tercera generación en el uso de Internet está caracterizada por una mayor interconexión y movilidad entre los usuarios: diferentes dispositivos entre los usuarios. El Web a partir de entonces no solo va a dar datos, también servicios programables o servicios Web (*Web Services*). Internet es el contexto de programación.

Pregunta: ¿Qué nos ofrece entonces la plataforma .NET?

> .NET es un ejemplo de integración uniforme y sin parches. En esta nueva plataforma de Microsoft la uniformidad y la consistencia se garantizan porque empieza desde cero. Bastaría entonces elegir un lenguaje .NET (por ejemplo, C#), y aprender ASP .NET y XML; el mismo lenguaje escogido se usaría para programar la lógica de la aplicación, validar en el servidor la interacción Web con cada cliente y hacer la GUI sobre Internet.

.NET es una plataforma informática que simplifica el desarrollo de aplicaciones en un entorno altamente distribuido como es Internet. Según [10], la creación de aplicaciones con .NET también brinda muchas mejoras no disponibles en otros lenguajes, como la seguridad. Estas medidas de seguridad pueden determinar si una aplicación puede leer o escribir archivos en un disco. También permite insertar firmas digitales en la aplicación para asegurarse que la misma fue escrita por una misma fuente de confianza. NET Framework también permite incluir información de componentes y de versión, dentro del código real. Esto hace posible que el software se instale cuando se lo pidan automáticamente o sin la intervención del usuario. Con .NET además disponemos de componentes reusables y autodocumentados, diversos lenguajes generando un mismo lenguaje intermedio y de una herramienta de desarrollo integrada a la Web.

V.1.2 .NET Framework

Según Gonzales Seco [12], "Microsoft.NET es el conjunto de nuevas tecnologías en las que Microsoft ha estado trabajando durante los últimos años con el objetivo de obtener una plataforma sencilla y potente para distribuir el software en forma de servicios que puedan ser suministrados remotamente y que puedan comunicarse y combinarse unos con otros de manera totalmente independiente de la plataforma, lenguaje de programación y modelo de componentes con los que hayan sido desarrollados. Esta es la llamada *plataforma .NET*, y a los servicios antes comentados se les denomina *servicios Web"*.

.NET fue diseñado con una idea en mente: potenciar el desarrollo de Internet. Este nuevo incentivo para el desarrollo se llama servicios Web. Puede pensar en los servicios Web como una página Web que interactúa con programas en lugar de con gente. En lugar de enviar páginas, un servicio Web recibe una solicitud en formato XML, realiza una función en concreto y luego devuelve una repuesta al solicitante en forma de mensaje XML. Un servicio Web no solo puede interactuar con páginas Web, puede interactuar con otros servicios Web [10].

La plataforma de desarrollo .NET se basa en 4 componentes principales que constituyen el .NET Framework y se muestran en la figura V.1.2.1 (en negrita se destacan los elementos que son de nuestro interés en este texto y que desarrollaremos en las secciones siguientes). Esto es lo que fundamenta que indistintamente denominemos a utilicemos .NET o .NET Framework.

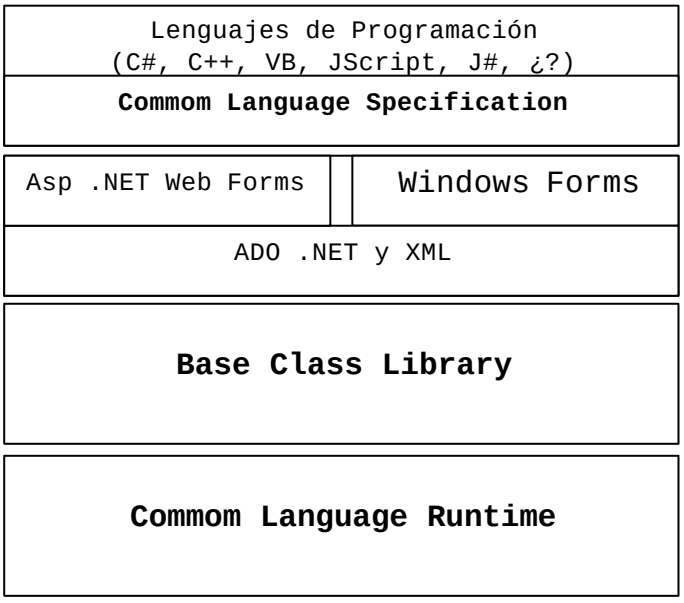


Figura V.1.2.1 Componentes principales de .NET Framework

Pregunta: ¿Cuáles son los objetivos de .NET Framework?

- Entorno coherente de POO, los objetos se pueden almacenar y ejecutar de forma local, ejecutar de forma local pero distribuida en Internet o ejecutar de forma remota.
- Reducir la implementación de software y los conflictos de versiones (infierno de las DLLs).
- Garantizar la ejecución segura del código.
- Coherencia entre tipos de aplicaciones muy diferentes, como las basadas en Windows o en el Web.
- Basar toda la comunicación en estándares para asegurar que el código de .NET Framework se puede integrar con otros tipos de código.
- Simplificar el desarrollo. Infraestructura de componentes (*assemblies*).
- Unificar los modelos de programación.
- Integración de los lenguajes.
- Usar los estándares Web y promocionar las buenas prácticas. Interoperabilidad.
- Confiabilidad y seguridad.

V.1.3 Entorno Común de Ejecucion (CLR)

Los lenguajes de programación generalmente se componen de un compilador y un entorno de ejecución. El compilador convierte el código fuente escrito por el usuario en código ejecutable que puede ser ejecutado por los usuarios. El entorno de ejecución proporciona al código ejecutable un conjunto de servicios del sistema operativo. Estos servicios están integrados en una capa de ejecución de modo que el código no necesite preocuparse de los detalles de bajo nivel de funcionamiento con el sistema operativo. Operaciones como la gestión de memoria y la entrada salida de archivos son buenos ejemplos de servicios realizados por un entorno de ejecución.

Antes de que se desarrollara .NET, cada lenguaje constaba de su propio entorno de ejecución. Los programadores entonces escribían código y luego lo compilaban con el propio entorno de ejecución en mente. El código ejecutable incluiría su propio entorno de ejecución. El principal problema que presentan estos entornos de ejecución es que estaban diseñados para usar un solo lenguaje.

Este enfoque de entornos de ejecución separados impedía que los lenguajes pudiesen funcionar conjuntamente sin problemas. Antes de .NET, los conjuntos de funciones de los diferentes entornos de ejecución eran inconsistentes.

Uno de los objetivos de diseño de .NET Framework era unificar los entornos de ejecución para que todos los programadores pudieran trabajar con un solo conjunto de servicios de ejecución. La solución de .NET Framework se llama *Entorno Común de Ejecución* (CLR *Common Language Runtime*). El CLR proporciona funciones para la gestión de la memoria, la seguridad y un sólido sistema de control de errores, a cualquier lenguaje que se integre en .NET Framework.

El CLR es el núcleo de la plataforma .NET. Es el motor encargado de gestionar la ejecución de las aplicaciones para ella desarrolladas. Por esta razón, al código de estas aplicaciones se le suele llamar *código gestionado.* A dichas aplicaciones el CLR le ofrece numerosos servicios que simplifican su desarrollo y favorecen su fiabilidad y seguridad.

Pregunta: ¿Cuáles son las principales características y servicios que ofrece el CLR?

Para responder a esta pregunta nos basamos en algunas ideas expuestas en [12] que a continuación se relacionan:

- **Modelo de programación consistente:** A todos los servicios y facilidades ofrecidos por el CLR se accede de la misma forma: a través de un modelo de programación basado en el paradigma OO.
- **Modelo de programación sencillo:** Con el CLR desaparecen muchos elementos complejos incluidos en los sistemas operativos actuales que dificultan el desarrollo de las aplicaciones (por ejemplo, registro de Windows).

- **Desaparece el "infierno de las DLLs":** En los sistemas operativos actuales de la familia Windows se produce un problema conocido como "infierno de las DLLs". Esto consiste en que al sustituirse versiones viejas de DLLs por versiones nuevas puede que las aplicaciones que fueron diseñadas para ser ejecutadas usando las viejas dejen de funcionar si las nuevas no son 100% compatibles con las anteriores. En la plataforma .NET desaparece este problema porque las versiones nuevas de las DLLs pueden coexistir con las viejas, de modo que las aplicaciones diseñadas para ejecutarse usando las viejas podrán seguir usándolas tras instalación de las nuevas. Por supuesto, esto simplifica mucho la instalación y desinstalación de software.
- **Ejecución multiplataforma:** El CLR actúa como una máquina virtual, encargándose de ejecutar las aplicaciones diseñadas para la plataforma .NET. Es decir, cualquier plataforma para la que exista una versión del CLR podrá ejecutar cualquier aplicación .NET. Microsoft ha desarrollado versiones del CLR para la mayoría de las versiones de Windows partiendo de Windows 95.
- **Interoperabilidad:** Gracias al CLR es posible escribir una clase en C# que use o amplíe las funcionalidades de otra escrita en Visual Basic.NET y viceversa. De forma general, desde cualquier lenguaje para el que exista un compilador que genere código para .NET es posible utilizar código .NET generado usando cualquier otro lenguaje tal y como si fuese código escrito usando el primero.
- **Gestión de memoria:** El CLR incluye un *recolector de basura* (*garbage collector*) que se ejecuta automáticamente y que evita que el programador tenga destruir los objetos que dejen de serle útiles. Este recolector de basura nos garantiza que no se produzcan errores de programación muy comunes como intentos de borrado de objetos ya eliminados, agotamiento de memoria por olvido de eliminación de objetos inútiles o solicitud de acceso a componentes de objetos ya destruidos.
- **Control de tipos:** El CLR es capaz de comprobar que toda conversión de tipos que se realice durante la ejecución de una aplicación .NET se haga de modo que los tipos origen y destino sean compatibles.
- **Tratamiento de excepciones:** En el CLR la propagación de los errores se realiza de forma homogénea: a través de excepciones y garantiza que excepciones lanzadas desde código para .NET escrito en un cierto lenguaje se puedan capturar en código escrito usando otro lenguaje (compatibles .NET por supuesto).

V.1.3.1 Lenguaje Intermedio de Microsoft (MSIL)

Pregunta: ¿Cuál es el fundamento de la interoperabilidad en los lenguajes .NET?

> El código generado por el compilador de C# está escrito en un lenguaje denominado *Lenguaje Intermedio de Microsoft* (MSIL-*Microsoft Intermediate Language*), con frecuencia apenas llamado IL. Este se compone de un conjunto específico de instrucciones que especifican como debe ser ejecutado el código. Contiene instrucciones para operaciones como la inicialización de las variables, llamadas a métodos, el control de errores, etc.

Todos los lenguajes compatibles con .NET producen MSIL cuando se compila su código fuente. Entonces, como todos los lenguajes .NET se compilan en el mismo conjunto de instrucciones MSIL y como todos los lenguajes .NET usan el mismo entorno de ejecución, los códigos de diferentes lenguajes y de diferentes compiladores pueden funcionar conjuntamente. Este es precisamente el fundamento de la interoperabilidad.

Todos los compiladores que generan código para la plataforma .NET no generan código de máquina para ningún CPU en concreto, sino que generan código MSIL. El CLR da a las aplicaciones la idea de que se están ejecutando sobre una máquina virtual, y precisamente MSIL es el código máquina de esa máquina virtual. Por tanto, MSIL es el único código que el CLR es capaz de interpretar. Es decir, cuando se dice que un compilador genera código para la plataforma .NET lo que se está diciendo es que genera MSIL [12].

Según González Seco [12], "MSIL ha sido creado por Microsoft tras consultar a numerosos especialistas en la escritura de compiladores y lenguajes tanto del mundo académico como empresarial. Es un lenguaje de un nivel de abstracción mucho más alto que el de la mayoría de los códigos máquina de las CPUs existentes, e incluye instrucciones que permiten trabajar directamente con objetos (crearlos, destruirlos, inicializarlos, llamar a métodos virtuales, etc.), tablas y excepciones (lanzarlas, capturarlas y tratarlas)".

Situación de análisis

MSIL no es un conjunto de instrucciones específicas para una CPU física, MSIL no sabe nada de su equipo y su equipo no conoce nada respecto a MSIL.

Pregunta: ¿Cómo se ejecuta entonces el código .NET, si su CPU no lo puede interpretar?

El código MSIL se convierte en código específico de CPU cuando se ejecuta por primera vez. Este proceso se llama Compilación *Justo a Tiempo* (JIT *Just in Time*). Este proceso de compilación es realizado por un compilador JIT que se encarga de convertir el código genérico IL en código específico que puede ser ejecutado en su CPU.

Pregunta: ¿Por qué generar MSIL cuando un compilador podría generar código específico para la CPU?

Esto es lo que han hecho los compiladores tradicionalmente. Sin embargo, IL permite que el código compilado se transfiera fácilmente a hardware diferente. El conjunto de instrucciones puede leerse fácilmente en un proceso de verificación. El proceso de verificación asegura que el código accede a la memoria correctamente y de que está usando los tipos de variables correctos al llamar a métodos que esperan un tipo específico, etc. Estos procesos de verificación aseguran que el código no ejecute ninguna instrucción que origine fallo.

V.1.3.2 Metadatos

Como resultado de la compilación en .NET se distinguen dos tipos de *ficheros*: *ejecutables* (extensión `.exe`) y *bibliotecas de enlace dinámico* (extensión `.dll` generalmente). Ambos ficheros contienen definiciones de tipos de datos, y se diferencian entre ellos es que solamente los primeros tienen un método especial que sirve de punto de entrada a partir del que es posible ejecutar el código que contienen haciendo una llamada desde la línea de comandos del sistema operativo. Los ficheros `.exe` pueden ejecutarse en cualquiera de los diferentes sistemas operativos de la familia Windows para los que existe alguna versión del CLR.

Pregunta: ¿Los ficheros resultantes de la compilación en .NET contienen solamente código MSIL?

El contenido de un fichero compilado no es sólo MSIL, en este se distinguen también la *cabecera de CLR* y los *metadatos*.

La cabecera de CLR es un pequeño bloque de información que informa que se trata de un módulo gestionado e indica la versión del CLR que necesita, cuál es su firma digital, cuál es su punto de entrada (si es un ejecutable), etc.

Por su parte, los metadatos constituyen un conjunto de datos que se organizan en forma de tablas que almacenan información sobre los tipos definidos en el fichero compilado, los miembros de éstos y sobre cuáles son los tipos externos al fichero compilado a los que se les referencia en éste.

Es decir, el proceso de compilación también produce metadatos, lo que es una parte importante de la filosofía de cómo se comparte el código .NET. Los compiladores de los lenguajes .NET colocan metadatos en el código compilado junto al código MSIL generado. Estos metadatos describen con exactitud todas las clases que escribimos en el código fuente, esto incluye a su vez la descripción de todos los componentes de las referidas clases.

Toda la información necesaria para describir una clase en un código está situada en el metadato, lista para ser leída en otras aplicaciones.

V.1.3.3 Ensamblados

Para Ferguson et al [10], un *ensamblado* es u paquete de código MSIL y metadatos. Cuando utiliza un conjunto de clases en un ensamblado, está usando las clases como una unidad y las mismas comparten el mismo nivel de control de versión, información de seguridad y requisitos de activación.

Según Gonzalez Seco [12], un *ensamblado* es una agrupación lógica de uno o más ficheros compilados o ficheros de recursos (ficheros .GIF, .HTML, etc.) que se engloban bajo un nombre común. Todo ensamblado contiene un *manifiesto*, que son metadatos con información sobre las características del ensamblado.

Una aplicación puede acceder a información o código almacenados en un ensamblado sin tener información sobre cuál es el fichero en concreto donde se encuentran. Los ensamblados nos brindan la posibilidad de abstraernos de la ubicación física del código que ejecutemos o de los recursos que usemos.

Pregunta: ¿Existe alguna diferenciación entre los ensamblados?

Hay dos tipos de ensamblados: *ensamblados privados* y *ensamblados compartidos o globales*. En el proceso de construcción de un ensamblado, no hay necesidad de especificar cual es el tipo del mismo.

Los ensamblados privados se almacenan en el mismo directorio que la aplicación que los usa y sólo puede usarlos ésta, mientras que los compartidos se almacenan en un Caché de Ensamblado Global (GAC *Global Assembly Cache*) y pueden usarlos cualquiera que haya sido compilada referenciándolos.

Con un ensamblado privado, hace que el código esté disponible para una sola aplicación. Este se empaqueta como una DLL y se instala (copia) en el mismo directorio de la aplicación que lo está usando. Usando este tipo de ensamblado, la única aplicación que puede usar el código es el ejecutable situado en el mismo directorio que el ensamblado.

Si nuestro objetivo es compartir el código con varias aplicaciones, entonces tenemos que recurrir al uso del código como ensamblado compartido o global. Estos pueden ser usados por cualquier aplicación .NET, sin tener en cuenta la localización de esta. Particularmente, Microsoft incorpora un conjunto de ensamblados en .NET Framework que se instalan como ensamblados globales.

También para evitar problemas, en el GAC se pueden mantener múltiples versiones de un mismo ensamblado. Así, si una aplicación fue compilada usando una cierta versión de un determinado ensamblado compartido, cuando se ejecute sólo podrá hacer uso de esa versión del ensamblado y no de alguna otra más moderna que se hubiese instalado en el GAC. De esta forma se soluciona el referido problema del infierno de las DLLs.

V.1.3.4 Sistema Común de Tipos (CTS)

El CTS (*Common Type System*) es el conjunto de reglas que han de seguir las definiciones de tipos de datos para que el CLR las acepte. Es decir, aunque cada lenguaje definido disponga de su propia sintaxis para definir tipos de datos, en el MSIL resultante de la compilación de sus códigos fuente se ha de cumplir las reglas del CTS. Algunos ejemplos de estas reglas son:

- Cada tipo de dato puede constar de cero o más miembros. Cada uno de estos miembros puede ser una variable, un método, una propiedad o un evento.
- No puede haber herencia múltiple, y todo tipo de dato ha de heredar directa o indirectamente de `System.Object`.

V.1.4 Biblioteca de Clases Base (BCL)

A los programadores les trasmite mucha confianza trabajar con código que ya ha sido probado y que funciona. La reutilización del código lleva mucho tiempo siendo un objetivo de la comunidad de desarrolladores de software. Sin embargo, la posibilidad de reutilizar el código no ha estado a la altura de las expectativas.

Muchos han sido los intentos por diseñar bibliotecas de clases o de funciones con el objetivo de brindar las más diversas funcionalidades o servicios. Sin embargo, la naturaleza específica del lenguaje de estas bibliotecas las ha hecho inservibles para ser usadas en otros lenguajes.

`.NET Framework` proporciona muchas clases que ayudan a los programadores a reutilizar el código. A través de las clases suministradas por .NET es posible desarrollar cualquier tipo de aplicación, desde las ya conocidas (y realizadas) aplicaciones consola y las tradicionales aplicaciones de ventanas hasta los novedosos servicios `Web` y páginas `ASP.NET`.

Pregunta: ¿Cuál fue la primera clase de C# (excluyendo los tipos de datos básicos) que utilizamos en el presente texto?

La clase `Math` que fue utilizada en I.2.3.1 para obtener el valor de la constante matemática π (`Math.PI`) en la expresión para el cálculo del área de la clase `Circunferencia`.

Pregunta: Mencione otras de las clases de C# que hemos utilizado hasta el momento

Sin lugar a dudas que la clase más utilizada hasta el momento es la clase `Console` para hacer uso de sus métodos `WriteLine` y `ReadLine` para realizar entrada/salida en modo consola. Otras clases utilizadas hasta el momento son: `DateTime` y `Environment`.

Pregunta: ¿Dónde están contenidas estas clases?

Ya sabemos que estas primeras clases que hemos utilizado hasta el momento están contenidas en el espacio de nombres (`namespace`) `System`, pero a su vez este *namespace* esta contenido en la Biblioteca de Clases Base (BCL *Base Class Library*) de `.NET`.

La BCL de `.NET` está formada por cientos de tipos de datos que permiten acceder a los servicios ofrecidos por el CLR y a las funcionalidades más frecuentemente usadas a la hora de escribir aplicaciones. Además, a partir de estas clases el programador puede crear nuevas clases que extiendan o usen su funcionalidad y se integren a la perfección con el resto de las clases de la BCL. Es tal la riqueza de servicios que ofrece esta biblioteca de clases que pueden crearse lenguajes que carezcan de biblioteca de clases propia y sólo usen la BCL (por ejemplo, C#) [12]; es por ello que cuando hablamos de una clase de C#, realmente estamos hablando de una clase de la BCL.

Dado la amplitud de la BCL, ha sido necesario organizar las clases en ella incluida en espacios de nombres que agrupen clases con funcionalidades similares. Precisamente es conocido por nosotros el espacio de nombres `System` que nos proporciona tipos que son usados muy frecuentemente, como los tipos de datos básicos, entrada/salida en consola, excepciones, tablas, etc. A su vez el espacio de nombres `System` contiene otros espacios de nombres de uso frecuente y que estudiaremos principalmente en las partes II y II del presente texto: `Collections` (colecciones de datos de uso común como pilas, colas, listas, diccionarios, etc.), `IO` (manipulación de ficheros y otros flujos de datos), etc.

Situación de análisis

Toda la BCL está disponible para cualquier lenguaje de programación compatible con `.NET`. Gracias al CLR, cualquier lenguaje .NET puede usar cualquier clase de la biblioteca .NET.

Pregunta: ¿Qué fundamenta el planteamiento de la situación de análisis anterior?

Esto se fundamenta en que la BCL está escrita en código MSIL, por lo que puede usarse desde cualquier lenguaje cuyo compilador genere MSIL y esto último es una condición necesaria para los lenguajes compatibles con `.NET`.

V.1.5 Especificacion del Lenguaje Común (CLS)

Situación de análisis

Frecuentemente hemos hecho referencia a la compatibilidad entre los lenguajes .NET y hasta ahora solamente se ha tenido en cuenta la generación de código MSIL.

Pregunta: ¿Existen otros elementos que garanticen la compatibilidad entre los lenguajes .NET?

Efectivamente, existen otras condiciones necesarias para garantizar la compatibilidad entre los lenguajes .NET. La Especificación del Lenguaje Común (CLS *Common Language Specification*) es un conjunto de reglas que han de seguir las definiciones de tipos que se hagan usando un determinado lenguaje .NET si se desea que sean accesibles desde cualquier otro lenguaje .NET. Obviamente, sólo es necesario seguir estas reglas en las definiciones de tipos y miembros que sean accesibles externamente, y no en las de los privados (II.3). Además, si no importa la interoperabilidad entre lenguajes tampoco es necesario seguirlas.

A continuación se listan algunas de reglas significativas del CLS:

- Los tipos de datos básicos admitidos son **`bool`**, **`char`**, **`byte`**, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**, **`string`** y **`object`**. Nótese pues que no todos los lenguajes tienen porqué admitir los tipos básicos enteros sin signo o el tipo **`decimal`** como lo hace C#.

- Las tablas han de tener una o más dimensiones, y el número de dimensiones de cada tabla ha de ser fijo. Además, han de indexarse empezando a contar desde 0 (Tema III).
- Se pueden definir tipos abstractos y tipos sellados. Los tipos sellados no pueden tener miembros abstractos (Parte II).
- Las excepciones han de derivar de `System.Exception` (II.3), las enumeraciones de `System.Enum`, (III.1), etc.
- En un mismo ámbito no se pueden definir varios identificadores cuyos nombres sólo difieran en la capitalización usada. De este modo se evitan problemas al acceder a ellos usando lenguajes no sensibles a mayúsculas.
- Las enumeraciones no pueden implementar interfaces, y todos sus campos han de ser estáticos y del mismo tipo. El tipo de los campos de una enumeración sólo puede ser uno de estos cuatro tipos básicos: **`byte`**, **`short`**, **`int`** o **`long`**.

V.1.6 Bibliografía complementaria

- Jose A. González Seco, El lenguaje de programación C#, 2002. Puede descargarse en http://www.josanguapo.com/

Referencias

[1] **T. Budd,** *Introducción a la Programación Orientada a Objetos*, Addison-Wesley Iberoamericana, 1994.

[2] **R. Cadenhead et al**, *Programación Aprendiendo Java 2.0 en 21 Días*, Pearson Spanish, 3003. ISBN 9701702298

[3] **M. Katrib**, *Programación Orientada a Objetos en C++*, X view, 1997.

[4] **J. Rumbaugh et al,** *Modelagem e Projetos Baseados en Objetos*, Campus Ltda, 1994. ISBN 85-7001-841-X.

[5] **C. Larman,** *UML y Patrones. Introducción al análisis y diseño orientado a objetos*, Prentice may, 1999, ISBN 970-17-0261-1.

[6] **J. García Molina, et al.**, *Una propuesta para organizar la enseñanza de la Orientación a Objetos*, VIII Jornadas de Enseñanza Universitaria de la Informatica (JENUI'2002), Cáceres, Julio, 2002.

[7] **G. Booch,** *Object-Oriented Analysis and Design with applications*, Addison-Wesley Longman, Inc, 1999. ISBN 0-8053-340-2.

[8] **M. Katrib, E. Quesada;** *Programación con Pascal*, Pueblo y Educación, 1991.

[9] **J. García Molina et al.**, *Introducción a la Programación*, ICE, Universidad de Murcia, Diego Marín, 1999.

[10] **J. Ferguson et al.**, *La Biblia de C#*, Anaya Multimedia, 2003. ISBN 84-415-1484-4.

[11] **J. García Molina,** *¿Es conveniente la orientación a objetos en un primer curso de programación?*, VII Jornadas de Enseñanza Universitaria de la Informática (JENUI'2001), Palma de Mallorca, 16-18 de Julio, 2001. Puede descargarse en http://dis.um.es/~jmolina/publi.html

[12] **J. A. González Seco,** *El lenguaje de programación C#*, 2001. Puede descargarse en http://www.josanguapo.com/

[13] **T. Archer;** *A Fondo C#*, McGraw-Hill/Interamericana. 2001. ISBN 84-481-3246-1.

[14] **B. Meyer;** *Object-Oriented Software Construction*, Prentice Hall. 1997. ISBN 0-13-629155-4.

[15] **J. Dockx et al,** *A new Pedagogy for Programming*, ECOOP'02, Educator's Symposium, 1996. Puede descargarse en http://prog.vub.ac.be/ecoop2002/ws03/acc_papers/Jan_Dockx.pdf

[16] **R. Pressman;** *Ingeniería del software: un enfoque práctico*, Mc.Graw-Hill. 1997.

[17] **B. Stroustrup;** *The C++ Programming Language*, Addison-Wesley Longman, Inc. 1997. ISBN 0-201-88955-4.

[18] **T. Quatrani;** *The Visual Modelling with Rational Rose 2000 and UML*, Addison-Wesley Longman, Inc. 2000. ISBN 0201699613.

[19] **ACM/IEEE,** *Computing Curricula 2001*, Final Report, December 2001, http://wwwcomputer.org/education/cc2001/final/index.htm

[20] **J. Bergin,** *Why Procedural is the Wrong First Paradigm if OOP is the Goal*, OOPSLA99, Educators' Symposium, http://www.acm.org/sigplan/oopsla/oopsla99/2_ap/2f_edusym.html, 1999. Puede descargarse en http://csis.pace.edu/~bergin/papers/Whynotproceduralfirst.html.

[21] **F. Charte;** *Visual C# .NET*, Anaya Multimedia. 2002. ISBN 84-415-1392-9.

[22] **G. Arora;** *C# .NET*, Anaya Multimedia. 2002. ISBN 84-415-1420-9.

[23] **C. Wille;** *C#*, Prentice Hall. 2001. ISBN 0-672-32037-1.

[24] **M. Katrib et al;***Plataforma .NET y Web Services*. Universidad de la Habana. 2003.

[25] **A. Hejlsberg, S. Wiltamuth;** *C# Language Reference*. Microsoft Corporation. 2000.

[26] **G. Booch, et al;***El Lenguaje Unificado para el Modelado.* Addison Wesley. 2000.