UNIVERSIDAD NACIONAL AUTONOMA DE NICARAGUA, (UNAN-LEON)

Tesis de Maestría en Computación con Énfasis en Gestión de Información

PLAN DOCENTE PARA LA ASIGNATURA DE PROGRAMACION ORIENTADA A OBJETOS

Autor: Licda. Karla María Mejía Ortiz

Tutor: MSc. Álvaro Rafael Altamirano Osorio

Julio, 2008

INDICE GENERAL

Capítulo 1: ASPECTOS INTRODUCTORIOS. *6*

1.1 INTRODUCCION **7**

1.2 OBJETIVOS **8**

1.3 DESARROLLO DEL PROYECTO **9**

Capítulo 2: PLAN DE ESTUDIOS Y METODOLOGIA DOCENTE. *10*

2.1 PROGRAMACION ORIENTADA A OBJETOS EN EL PLAN DE ESTUDIOS EN LA CARRERA DE LICENCIATURA EN ESTADISTICA. **11**

2.1.1 Asignatura: Introducción a la Informática. 12

2.1.2 Asignatura: Programación I. 13

2.1.3 Asignatura: Programación II. 14

2.1.4 Plan de estudios de la Carrera Licenciatura en Estadística. 15

2.2 METODOLOGIA DOCENTE **17**

2.2.1 Material y Método Didáctico 17

2.2.2 Metodología de Evaluación 18

Capitulo 3: PROGRAMACION ORIENTADA A OBJETOS (POO). *20*

3.1 INTRODUCCION **21**

3.1.1 Objetivos 22

3.1.2 Planificación Temporal 23

3.1.3 Contenido del Temario Teórico 24

3.2 PRESENTACION DE LA ASIGNATURA **26**

3.3 UNIDAD I: C# Y LA PLATAFORMA .NET **28**

3.3.1 Introducción 28

3.3.2 Microsoft .NET 28

3.3.3 El lenguaje C# 29

3.3.4 Instalación del Entorno de Desarrollo 30

3.4 UNIDAD II: INTRODUCCION A LA PROGRAMACION ORIENTADA A OBJETOS **33**

3.4.1 Programación Orientada a Objetos (POO) 33

3.4.2 Mecanismos básicos de la POO 33

3.4.3 Características de la POO 34

3.4.4 Lenguajes orientados a objetos 35

3.4.5 Ventajas de los lenguajes orientados a objetos 35

3.5 UNIDAD III: CLASES **37**

3.5.1 Definición de una clase 37

3.5.2 Miembros de una clase 38

3.5.3 Control de acceso a los miembros de la clase 39

3.5.4 Implementación de una clase: CFecha 40

3.5.5 Métodos sobrecargados 42

3.5.6 Referencia this 44

3.5.7 Construcción de objetos 44

3.5.8 Asignación de objetos 45

3.5.9 Constructor copia 46

3.5.10 Destrucción de objetos 47

3.6 UNIDAD IV: OPERADORES SOBRECARGADOS 50
3.6.1 Sobrecargar un operador 50
3.6.2 Ejemplo: la clase CComplejo 52
3.6.3 Sobrecarga de operadores binarios 54
3.6.4 Sobrecarga de operadores de asignación y operadores aritméticos 54
3.6.5 Aritmética mixta 57
3.6.6 Sobrecarga de operadores unarios 58
3.6.7 Operadores unarios/binarios 59
3.6.8 Conversiones personalizadas 59
3.6.9 Indexación 60

3.7 UNIDAD V: HERENCIA, POLIMORFISMO E INTERFACES 62
3.7.1 Concepto de herencia 62
3.7.2 Definir una clase derivada 64
3.7.3 Control de acceso a los miembros de las clases 65
3.7.4 Atributos con el mismo nombre 66
3.7.5 Redefinir métodos de la clase base 68
3.7.6 Constructores de las clases derivadas 69
3.7.7 Copia de objetos 70
3.7.8 Destructores de las clases derivadas 71
3.7.9 Jerarquía de clases 72
3.7.10 Métodos virtuales 72
3.7.11 Polimorfismo 75
3.7.12 Clases y métodos abstractos 77
3.7.13 Interfaces 79
3.7.14 Interfaces frente a herencia múltiple 81

3.8 UNIDAD VI: EXCEPCIONES 84
3.8.1 ¿Qué son las excepciones? 84
3.8.2 Excepciones de C# 84
3.8.3 Manejar excepciones 85
3.8.4 Lanzar una excepción 85
3.8.5 Capturar una excepción 86
3.8.6 Excepciones derivadas 87
3.8.7 Capturar cualquier excepción 87
3.8.8 Relanzar una excepción 87
3.8.9 Bloque de finalización 87
3.8.10 Crear excepciones 88

Capitulo 4: LABORATORIO DE PROGRAMACION ORIENTADA A OBJETOS.... 90

4.1 INTRODUCCION 91
4.1.1 Objetivos 92
4.1.2 Planificación Temporal. 93

PRÁCTICA 1: FUNDAMENTOS DE C#, TRABAJANDO CON MATRICES 94
4.1.3 Enunciado de la práctica 94

4.2 PRÁCTICA 2: CREACIÓN Y DEFINICIÓN DE UNA CLASE PARA EL MANEJO DE HORAS 95
4.2.1 Enunciado de la práctica 95

4.3 PRÁCTICA 3: OPERADORES SOBRECARGADOS UTILIZANDO LA CLASE CHORA. 100
4.3.1 Enunciado de la práctica 100

4.4 PRÁCTICA 4: DERIVACIÓN DE LA CLASE CENFERMO Y CEMPLEADO TOMANDO COMO CLASE BASE LA CLASE CFICHA DE LA PRÁCTICA 2. IMPLEMENTACIÓN DE LA CLASE CHOSPITAL. 101
4.4.1 Enunciado de la práctica.....101
4.5 PRÁCTICA 5: EXCEPCIONES. 104
4.5.1 Enunciado de la práctica.....104
BIBLIOGRAFÍA..... 105
ANEXOS..... 106
Solución de la práctica 1 107
Solución de la práctica 2 111
Solución de la práctica 3 117
Solución de la práctica 4 122
Solución de la práctica 5 132

PLAN DOCENTE DE LA ASIGNATURA:

PROGRAMACION ORIENTADA A OBJETOS

Capítulo 1: ASPECTOS INTRODUCTORIOS.

1.1 INTRODUCCION

La Facultad Multidisciplinaria Oriental de la Universidad de El Salvador, cuenta con la carrera de Licenciatura en Estadística la cual busca dar una contribución a una serie de necesidades de carácter educativo, científico y productivo que actualmente vive El Salvador, ésta carrera conjuga las áreas de estadística, matemática y computación, que son disciplinas importantes para la etapa de desarrollo social y económico del país. En la carrera de Licenciatura en Estadística se sirven asignaturas electivas, las cuales se conciben de manera flexible y dinámica con el propósito de tener en cuenta la acelerada producción de nuevo conocimiento y al mismo tiempo poder aprovechar al máximo la oferta de las diferentes unidades académicas de la Universidad. De tal forma que cada año académico el Consejo Asesor de Planificación Académica de la Escuela de Matemática define las asignaturas electivas que son ofertadas a los estudiantes de los diferentes niveles. Desde ya hace tres años en la facultad se imparte la asignatura de Programación Orientada a Objetos con Java.

En el presente documento se desarrolla un plan docente para la asignatura electiva "**Programación Orientada a Objetos**" con el lenguaje orientado a objetos C#, que será impartida en la carrera de Licenciatura en Estadística de la Facultad Multidisciplinaria Oriental de la Universidad de El Salvador, cuando ésta asignatura sea ofertada.

1.2 OBJETIVOS

- Realizar un proyecto docente que contenga los aspectos básicos de la Programación Orientada a Objetos.
- Desarrollar el material necesario, teórico y práctico de la asignatura Programación Orientada a Objetos.
- Elaborar una herramienta para el docente encargado de impartir la asignatura así como el alumno que la recibe.
- Planificar en la asignatura la parte teórica y práctica con la metodología a seguir y las evaluaciones a realizar.
- Que el alumno aprenda y maneje la metodología de la programación orientada a objetos, tomando en cuenta sus conceptos y aplicación en solución de problemas con el lenguaje C#.

1.3 DESARROLLO DEL PROYECTO

En la realización de este trabajo se han seguido varias fases, se muestra las asignaturas de la carrera de Licenciatura en Estadística que son el preámbulo para entrar a la Programación Orientada a Objetos, se establece todo el material didáctico, la planificación del tiempo que tomara desarrollar los contenidos teóricos y prácticos, la metodología a emplear y el contenido de cada uno de los temas a tratar. En el desarrollo cada uno de los temas de la asignatura se establece objetivos, contenido de subtemas, duración y la bibliografía. Se desarrollan las prácticas que serán empleadas en el Laboratorio y sus soluciones.

En el desarrollo del contenido teórico, para mostrar los ejemplos y el contenido práctico se hizo uso el entorno de desarrollo SharpDevelop 2.2 y el paquete Microsoft .NET Framework Software Development Kit (SDK), los cuales se pueden obtener de forma gratuita de Internet.

Capítulo 2: PLAN DE ESTUDIOS Y METODOLOGIA DOCENTE.

2.1 PROGRAMACION ORIENTADA A OBJETOS EN EL PLAN DE ESTUDIOS EN LA CARRERA DE LICENCIATURA EN ESTADISTICA.

El estudiante que curse la asignatura de Programación Orientada a Objetos, debe tener conocimientos previos de programación en un Lenguaje estructurado, para que pueda aprender los conceptos y la metodología en que se basa la Programación Orientada a Objetos.

En el plan de estudios de la carrera de Licenciatura en Estadística, desde el ciclo II primer año de la carrera se imparte la asignatura Introducción a la Informática, luego Programación I en el ciclo III del segundo año de la carrera, que para cursar Programación I es prerrequisito haber aprobado Introducción a la Informática, al haber aprobado Programación I el estudiante puede inscribir Programación II, ciclo IV del segundo año de la carrera; después de estas se imparte una electiva, que ha sido desde varios años la Electiva I: Programación Orientada a Objetos.

En las asignaturas de Programación I y Programación II se trabaja con el lenguaje C.

A continuación se muestra los silabos de las asignaturas antes mencionadas.

2.1.1 Asignatura: Introducción a la Informática.

I. INTRODUCCION A LA INFORMATICA
II. GENERALIDADES:

Código: INI1109	Prerrequisito (s): Lógica Matemática.	U.V. : 4	Nivel: Ciclo II
Duración: 17 semanas, 96 Horas-clase	Horas Teóricas: 64		Horas Prácticas: 32

III. DESCRIPCION SINTÉTICA.
El curso pretende que el estudiante sea capaz de elaborar soluciones eficientes y que el computador pueda ejecutarlas . Para ello se estudiará el funcionamiento del computador, el manejo de los datos, los algoritmos, el diseño descendente y estructurado, y técnicas de representación de algoritmos.

IV. OBJETIVOS.
1. Que el estudiante comprenda qué es un computador, como funciona y para qué sirve.
2. Que el estudiante sepa cómo diseñar estructurada y adecuadamente soluciones algorítmicas de problemas propios de su competencia.
3. Que el estudiante aprenda y aplique estructuras de datos y técnicas de programación que le permitan diseñar algoritmos y por ende programas eficientes, seguros y funcionales.

V. CONTENIDO.
- Introducción al computador y sus fundamentos.
- Sistemas de numeración y representación de la información.
- Proceso para resolver un problema utilizando el computador.
- Algoritmos y propiedades.
- Manejo de constantes y variables.
- Operaciones básicas (asignación de memoria, ent/sal. De datos, operaciones aritméticas).
- Algoritmos simples o con secuenciación.
- Problemas y algoritmos con estructuras de transferencia del control.
- Problemas y algoritmos con arreglos.
- Problemas y algoritmos particionados (funciones o subrutinas).

NOTA: Para una mejor estructuración y comprensión de los algoritmos, se utilizarán métodos de representación tales como: Pseudocódigo, diagramas de flujo, diagramas de Chapín y diagramas de Warnier.

VI. BIBLIOGRAFÍA.
1. ALCALDE, EDUARDO , GARCÍA, MIGUEL. metodología de la programación. Aplicaciones en Cobol y en Pascal. Editorial McGraw-Hill.
2. LÓPEZ, LEOBALDO, ROMÁN. Programación Estructurada. un enfoque algorítmico. Computec Editores, S.A. de C.V.

Material de apoyo. Módulo 1. FUNDAMENTOS DE ARQUITECTURA DEL COMPUTADOR. (Maestría en Informática – UES).

2.1.2 Asignatura: Programación I.

I. PROGRAMACION I
II. GENERALIDADES:

Código: PRO1109 Prerrequisito (s): Introducción a la Informática. U.V.:4 Nivel: Ciclo III
Duración: 17 semanas, 96 Horas-clase Horas Teóricas: 64 Horas Prácticas: 32

III. DESCRIPCION SINTÉTICA.

El curso pretende que el estudiante pueda resolver problemas propios de su disciplina, utilizando el computador, y creando para ello programas en lenguaje C que estén bien estructurados, que sean funcionales y que su documentación facilite el mantenimiento de los mismos.

IV. OBJETIVOS.

1. Aplicar las técnicas de programación estructurada, la modularidad y la programación en lenguaje C, para implementar soluciones algorítmicas robustas, eficientes y confiables.
2. Que el estudiante sepa resolver problemas complejos, mediante su partición en subprogramas o funciones.
3. Que el estudiante sea capaz de escribir y verificar sus programas utilizando un compilador adecuado (Turbo C++ de Borland ó Borland C++).

V. CONTENIDO.

- Introducción al lenguaje C.
- Tipos de datos y su declaración.
- Entrada / salida estándar de datos en el lenguaje C.
- Operadores en el lenguaje C.
- Funciones incorporadas por el fabricante.
- Estructuras de transferencias de control.
- Manejo de arreglos en C.
- Creación de funciones diseñadas por el usuario.
- Recursividad.
- Manejo de archivos en el lenguaje C.

VI. BIBLIOGRAFÍA.

1. GOTTFRIED, BYRON S. Programación en C. Editorial McGraw – Hill.
2. SCHILD, HERBERT. Aplique turbo C++. Editorial Osborne McGraw – Hill.
3. Material de apoyo. MODULO 4, INTRODUCCIÓN A LA PROGRAMACIÓN (Maestría en Informática).

2.1.3 Asignatura: Programación II.

I. PROGRAMACION II
II. GENERALIDADES:

Código: PRO2109	Prerrequisito (s): Programación I. U.V.: 4	Nivel: Ciclo IV.
Duración: 17 semanas, 96 Horas-clase	Horas Teóricas: 64	Horas Prácticas: 32

III. DESCRIPCION SINTÉTICA.
El curso busca estudiar los principios y elementos fundamentales de los dos principales enfoques usados actualmente en el diseño lógico y físico de programas. En el primer enfoque conocido como diseño procedimental se abordarán las estructuras estáticas (registros), punteros, asignación dinámica de memoria y estructuras dinámicas más el diseño descendente y programación modular.
El segundo enfoque del diseño y programación orientada a objetos, comprende los fundamentos del lenguaje C++, la filosofía del diseño orientado a objetos, las propiedades de las clases y objetos y algunas aplicaciones prácticas.

IV. OBJETIVOS.
1. Que el estudiante aprenda y aplique: el ciclo de vida de un proyecto, otras estructuras de datos estáticos como los registros, estructuras dinámicas más complejas como las pistas y árboles, técnicas de diseño descendente – modular.
2. Que el estudiante aprenda y aplique las características de los elementos del diseño orientado a objetos (clases y objetos), tales como: la abstracción, el encapsulamiento, la herencia y el polimorfismo.
3. Que el estudiante pueda desarrollar aplicaciones reales utilizando un compilador como el Turbo C++, o Visual C++.

V. CONTENIDO.
- Introducción al diseño de programas.
- Registros o estructuras en lenguaje C ó C++.
- Manejo de punteros o apuntadores en C ó C++.
- Asignación dinámica de memoria.
- Estructuras dinámicas de datos en C.
- Fundamentos del C++.
- Programación orientada a objetos.
- Objetos.
- Herencia entre clases.
- Polimorfismos.

VI. BIBLIOGRAFÍA.

- DEITEL, H:M:, DEITEL P.J.. C++ Como programar. Editorial Prentice Hall.
- SCHILD, HERBERT. .Aplique turbo C++. Editorial Osborne McGraw Hill.
- CEVALLOS, FRANCISCO JAVIER. Curso de programación con C. Editorial Macrobit.
- KRUSE L., ROBERT. Estructuras de datos y diseño de programas. Editorial Prentice Hall.

2.1.4 Plan de estudios de la Carrera Licenciatura en Estadística.

PLAN TIPO - LICENCIATURA EN ESTADISTICA

CICLO I	4 1 Matemática Básica	4 2 Lógica Matemática	4 3 Geometría I	4 4 Humanística I	
CICLO II	4 5 Cálculo I 1	4 6 Algebra Lineal I	4 7 Introducción a la Matemática Discreta 2	4 8 Introducción a la Informática 2	4 9 Humanística II *
CICLO III	4 10 Cálculo II 5	4 11 Algebra Lineal II 6	4 12 Estadística I 5	4 13 Programación I 8	
CICLO IV	4 14 Cálculo III 10	4 15 Estadística de Población I 12	4 16 Estadística II 10,12	4 17 Programación II 13	
CICLO V	4 18 Cálculo IV 14	4 19 Ecuaciones Diferenciales I 14	4 20 Inferencia Estadística 16	4 21 Electiva I *	
CICLO VI	4 22 Modelos Lineales 11,20	4 23 Sistemas de Redes de Computadoras 17	4 24 Muestreo I 20	4 25 Electiva II *	
CICLO VII	4 26 Análisis Numérico I 13,14	4 27 Métodos de Optimización I 11,13	4 28 Metodología de la Investigación 22,24	4 29 Electiva III *	
CICLO VIII	4 30 Análisis Multivariante I 11,20	4 31 Series Temporales I 22	4 32 Diseño de Experimentos I 11,20	4 33 Base de Datos 11,17	
CICLO IX	4 34 Análisis de Sistemas 33	4 35 Control Estadístico de la Calidad I 32	4 36 Electiva IV *	4 37 Seminario I U.V. 132	
CICLO X	4 38 Proyectos de Estudios Estadísticos 37	4 39 Electiva V *	4 40 Seminario II 37		

* = Según corresponda.

UV ORD
Asignatura
PRERREQ.

Como pudo observarse en el silabo de la asignatura de Programación II el estudiante aprende sobre la Programación Orientada a Objetos, Clase, Herencia y Polimorfismo aunque no se profundiza tanto en el tema. Para esta asignatura, Programación Orientada a Objetos el alumno tiene los conocimientos necesarios para estudiarla.

El Objetivo es que el estudiante se interese en la Programación en C#, y la Programación Orientada a Objetos y modele en la computadora problemas prácticos de la vida real, utilizando en lenguaje de programación C#.

2.2 METODOLOGIA DOCENTE

2.2.1 Material y Método Didáctico

Un ciclo académico tiene una duración de 16 semanas, y la asignatura Electiva I tiene 4 unidades valorativas que corresponden a tres horas semanales de clase teórica y dos horas semanales de laboratorio; lo que equivale a 48 horas de clase teórica y 32 horas de prácticas de laboratorio en todo el ciclo.

En esta asignatura de Programación Orientada a Objetos la metodología a emplear consiste en exposición de las lecciones correspondientes a los temas desarrollados en este plan en el aula por parte del docente durante las tres horas semanales de clase teórica tomando en cuenta la temporización que se detalla en cada uno de los temas, incluyendo la realización de ejercicios y actividades de enseñanza-aprendizaje que fomenten la formación de habilidades en los estudiantes. Discusión de temas relacionados con la asignatura permitiendo una retroalimentación estudiante-estudiante y estudiante-profesor.

El docente propondrá de una serie de prácticas de laboratorio que el alumno desarrollará para que logre un adecuado aprendizaje de la asignatura, estas prácticas también deben de ser desarrolladas tomando en cuenta el tiempo establecido para cada una de ellas.

Se hará la instalación en la plataforma Windows del software a utilizar en el laboratorio para el correspondiente desarrollo de las prácticas de la asignatura, el paquete Microsoft .NET Framework SDK 2.0 y el entorno de desarrollo integrado SharpDevelop 2.2.

También se establecerá un espacio de tutorías al inicio del ciclo académico, dos horas semanales, para que el estudiante pueda hacer consultas al docente sobre inquietudes con respecto a la asignatura, para que de esa forma pueda resolver dudas y puede complementar su autoestudio, además se proporcionará todo el material de manera impresa y también todo el material de apoyo empleado para el desarrollo de la asignatura, así como la bibliografía básica y complementaria de la misma.

Por parte de los estudiantes debe de haber actividades de aprendizaje independiente, en las que se preparen diversos temas para investigación relacionados con el programa curricular de la asignatura, fomentando en el estudiante el desarrollo de habilidades de investigación, comunicación, trabajo en equipo, entre otras, y la profundidad de conocimientos sobre temas orientados en clases que permitan una mayor comprensión y adquisición de conocimientos.

Los materiales didácticos para el desarrollo de la asignatura son el uso de pizarra acrílica, plumones, retroproyector. Y en algunos casos se hará uso de un ordenador para hacer pruebas de un tema especifico o mostrar alguna presentación donde se halla desarrollado un tema determinado, esto cuando el docente lo estime necesario.

2.2.2 Metodología de Evaluación

En la asignatura de Programación Orientada a Objetos se realizaran 3 exámenes parciales con una ponderación de 15% dos de ellos y con una ponderación de 20% el ultimo, teniendo así 50% de la nota final de ciclo. Se realizaran 2 exámenes prácticos con una ponderación de 10% y 15%, obteniendo 25% de la nota final de ciclo, y al final del ciclo un trabajo con una ponderación de 25%; completando de esa forma el 100% de la nota final de ciclo.

Como se desglosa a continuación:

Evaluación	Número	Ponderación
Parciales teóricos	**3**	**15%,15%, 20%**
Exámenes prácticos	**2**	**10%, 15%**
Trabajo final	**1**	**25%**
	TOTAL	**100%**

La escala de evaluación es de 0 a 10 siendo la nota mínima establecida para aprobar una evaluación de 6.0.

En las pruebas evaluadas teóricas se harán preguntas directas sobre los temas a evaluar previamente establecidas que formaran parte de la evaluación, para poder evaluar el grado de comprensión de los conceptos básicos que con respecto al tema tiene el estudiante. Incluirán preguntas de carácter analítico, las cuales se basen en un problema planteado en donde el estudiante tenga que realizar un análisis a partir de los conocimientos teóricos que haya adquirido.

En las evaluaciones prácticas se dará dos o más problemas para que el estudiante resuelva y aplique la teoría sobre el o los temas que se estén evaluando y otros conocimientos que el estudiante debe de tener para la solución del examen.

El trabajo de final de ciclo será un trabajo donde el estudiante pueda aplicar todos los conocimientos sobre la Programación Orientada a Objetos, problemas reales planteados para que el estudiante de acuerdo a la recopilación de su conocimiento resuelva de forma que funcione, explicando la solución, presentando pruebas de corridas, conclusiones y comentarios; el trabajo tendrá que ser defendido por cada estudiante; en este trabajo se pretende evaluar tantos los conocimientos adquiridos como el ingenio y la creatividad del estudiante para abordar un problema donde este presente los paradigmas de la Programación Orientada a Objetos.

Este trabajo se dejara con un tiempo prudencial para que los estudiantes trabajen en el mismo y que vayan aplicando la teoría vista durante el desarrollo de la asignatura; seis semanas previas a la finalización del ciclo, se dejará el trabajo y la entrega y defensa una semana antes de la finalización del ciclo académico.

Capitulo 3: PROGRAMACION ORIENTADA A OBJETOS (POO).

3.1 INTRODUCCION

La programación Orientada a Objetos es una asignatura electiva que esta siendo impartida en el ciclo V de la carrera de Licenciatura en Estadística, esta electiva tiene cuatro unidades valorativas (4UV), que corresponden a tres horas semanales de clases teóricas y dos horas semanales de laboratorio.

Para impartir esta asignatura se requiere que los estudiantes tengan conocimientos sobre la programación estructurada, algoritmos y estructuras de datos, los cuales son impartidos en las asignaturas previas a ésta, Introducción a la Informática, Programación I y Programación II.

Con el desarrollo de ésta asignatura se pretende introducir al estudiante en la metodología de la Programación Orientada a Objetos haciendo uso del lenguaje C# y que el estudiante sea motivado por este paradigma para que plantee y modele problemas reales en un ordenador.

En este plan docente se plantea la parte teórica y práctica sobre la **Programación Orientada a Objetos con C#** se tratan los conceptos bases de la OOP, mostrando al estudiante el modelo implementado con C#, con una serie de ejemplos y prácticas de laboratorio para que el estudiante afiance conceptos desde la creación de clases, hasta conceptos como herencia, polimorfismo, etc., propios de esta filosofía de trabajo.

En la primer tema se presenta una introducción a la plataforma .NET y lo que es C#, características de C#, en el segundo tema se hace una introducción a la Programación Orientada a Objetos, mostrando algunas características y ventajas de la POO, en el tema tercero se trata lo son las clases, aquí se presenta la definición de una clase y su implementación con una clase CFecha , en el tema cuarto la sobrecarga de operadores, en el tema quinto se trabaja con Herencia, Polimorfismo e Interfaces, y para finalizar en el tema sexto se trata las Excepciones.

3.1.1 Objetivos

Los objetivos de esta asignatura son:

- Que el estudiante adquiera los conocimientos relacionados con los aspectos básicos de la programación orientada a objetos.
- Capacitar al estudiante escritura de programas con el lenguaje de programación C# y aprovechar sus ventajas
- Desarrollar en el estudiante su capacidad de análisis, ingenio y creatividad a través del estudio y desarrollo de programas orientados a objetos a nivel de consola, con el entorno de desarrollo SharpDevelop 2.2. y en línea de ordenes con .NET Framework SDK 2.0.
- Que en los estudiantes haya la motivación para resolver problemas de la vida real haciendo uso del estilo de programación orientado a objetos.
- Construir programas donde se haga uso de la biblioteca de clase de .NET.

3.1.2 Planificación Temporal

Para el desarrollo adecuado de todo el contenido de la asignatura Programación Orientada a Objetos se ha planificado de acuerdo a la duración de un ciclo académico de la Universidad de El Salvador, el cual consta de 16 semanas. Y la asignatura electiva I tiene 4 unidades valorativas se dispone de 48 horas semanales para la parte teórica y 32 horas semanales para la parte practica, haciendo un total de 80 horas en todo el ciclo. De las horas de teoría el primer día de clases se tomará unos minutos para presentar a los estudiantes el programa de la asignatura, la metodología a seguir y el método de evaluación. De igual forma durante la primera practica se tomaran unos minutos para instalar en el ordenador lo básico para trabajar en las practicas, hacer la instalación del software a emplear.

La realización de los exámenes teóricos y prácticos se hará en otras horas que no sean de clase, ni de práctica, esto para que los estudiantes aprovechen las horas disponibles para afianzar los conocimientos de la asignatura. Se buscará con anticipación los espacios disponibles de tiempo de los estudiantes para programar las evaluaciones que tendrán una duración de 2 horas. También se verá con anticipación la disponibilidad del laboratorio para realizar los exámenes prácticos. En el siguiente cuadro se muestra la temporización para impartir la parte teórica de la asignatura:

UNIDAD	SEMANA	CONTENIDO	HORAS
-	1	Presentación de la asignatura	1
I	1-2	C# y la Plataforma .NET	2
II	3-4	Introducción a la Programación Orientada a Objetos	3
III	4-5-6-7	Clases	6
IV	8-9-10-11	Operadores sobrecargados	6
V	11-12-13-14	Herencia, Polimorfismo e interfaces	8
VI	15-16	Excepciones	6
		TOTAL DE HORAS	**48**

En este cuadro se presenta la temporización para la parte práctica de la asignatura:

PRACTICA	SEMANA	CONTENIDO	HORAS
1	1-2-3	Fundamentos de C#, Trabajando con Matrices.	6
2	4-5-6-7	Creación y definición de una Clase para el manejo de Horas.	8
3	8-9-10	Operadores Sobrecargados utilizando la clase CHora.	6
4	11-12-13-14	Derivación de la Clase **CEnfermo** y **CEmpleado** tomando como clase base la Clase **CFicha** de la Práctica 2. Implementación de la Clase **CHospital**.	8
5	15-16	Excepciones.	4
		TOTAL DE HORAS	**32**

3.1.3 Contenido del Temario Teórico

UNIDAD I: C# Y LA PLATAFORMA .NET

1. Introducción
2. Microsoft .NET
3. El Lenguaje C#
4. Instalación del Entorno de Desarrollo

UNIDAD II: INTRODUCCION A LA PROGRAMACION ORIENTADA A OBJETOS

1. Programación Orientada a Objetos (POO)
2. Mecanismos básicos de la POO
3. Características de la POO
4. Lenguajes Orientados a Objetos
5. Ventajas de los lenguajes Orientados a Objetos

UNIDAD III: CLASES

1. Definición de una clase
2. Miembros de una clase
3. Control de acceso a los miembros de la clase
4. Implementación de una clase: CFecha
5. Métodos sobrecargados
6. Referencia this
7. Construcción de objetos
8. Asignación de objetos
9. Constructor copia
10. Destrucción de objetos

UNIDAD IV: OPERADORES SOBRECARGADOS

1. Sobrecargar un operador
2. Ejemplo: la clase CComplejo
3. Sobrecarga de operadores binarios
4. Sobrecarga de operadores de asignación y operadores
5. aritméticos
6. Aritmética mixta

7. Sobrecarga de operadores unarios
8. Operadores unarios/binarios
9. Conversiones personalizadas
10. Indexación

UNIDAD V: HERENCIA, POLIMORFISMO E INTERFACES

1. Concepto de herencia
2. Definir una clase derivada
3. Control de acceso a los miembros de las clases
4. Atributos con el mismo nombre
5. Redefinir métodos de la clase base
6. Constructores de las clases derivadas
7. Copia de objetos
8. Destructores de las clases derivadas
9. Jerarquía de clases
10. Métodos virtuales
11. Polimorfismo
12. Clases y métodos abstractos
13. Interfaces
14. Interfaces frente a herencia múltiple

UNIDAD VI: EXCEPCIONES

1. ¿Qué son las excepciones?
2. Excepciones de C#
3. Manejar excepciones
4. Lanzar una excepción
5. Capturar una excepción
6. Excepciones derivadas
7. Capturar cualquier excepción
8. Relanzar una excepción
9. Bloque de finalización
10. Crear excepciones

3.2 PRESENTACION DE LA ASIGNATURA

UNIDAD N° 0

OBJETIVOS

- Dar a conocer al estudiante los objetivos generales planteados de la asignatura y los temas a desarrollar en cada una de las unidades.

- Conocer la metodología docente a desarrollar en la asignatura, así como los criterios de evaluación y la bibliografía necesaria.

- Dar a conocer al estudiante la planificación temporal de cada una de las unidades de la asignatura en todo el ciclo académico.

- Hacer del conocimiento al estudiante la disponibilidad de atender las preguntas o interrogantes que a través de las clases teóricas y prácticas aparezcan, dando también horarios de atención.

ASPECTOS IMPORTANTES

En esta unidad se muestra al estudiante todos los lineamientos a seguir en el desarrollo de la asignatura, es una unidad en la que se le da a conocer al estudiante la forma de trabajar durante el ciclo, las responsabilidades que el docente tendrá y las que él tendrá que cumplir para lograr un mejor aprovechamiento en la asimilación de todo el contenido temático de la asignatura.
Aquí se le da a conocer al estudiante como esta la distribución de tiempo en cada una de las unidades de la asignatura, se le hace ver que en la medida de lo posible se cumplirá con los tiempos distribuidos.
También se le muestra como va a ser el método de evaluación, la cantidad de evaluaciones a realizar, y el material al cual puede tener acceso en la sección de matemática, y la bibliografía que se utilizara para el desarrollo de la asignatura.

DURACION

1 Hora

3.3 UNIDAD I: C# Y LA PLATAFORMA .NET

Título del tema:

C# y la Plataforma .NET

Objetivo:

- Mostrar las partes de la que consta la plataforma .NET y las características del lenguaje de programación orientado a objetos C#.

Contenido:

- Introducción
- Microsoft .NET
- El Lenguaje C#
- Instalación del Entorno de Desarrollo

Duración:

- 2 horas

Bibliografía básica:

- Ceballos Sierra, Francisco Javier. Microsoft C#, Curso de Programación. Editorial RAMA.

3.3 UNIDAD I: C# Y LA PLATAFORMA .NET

3.3.1 Introducción

La plataforma .NET es una plataforma de desarrollo de software con especial énfasis en el desarrollo rápido de aplicaciones, la independencia de lenguaje y la transparencia a través de redes.

C# (C Sharp) es parte de la plataforma .NET. C# es un lenguaje orientado a objetos simple, seguro, moderno, de alto rendimiento y con especial énfasis en Internet y sus estándares (como XML). Es también la principal herramienta para programar en la plataforma .NET.

3.3.2 Microsoft .NET

¿Qué es la plataforma .NET?

Microsoft .NET Framework o .NET se trata de un entorno de desarrollo multilenguaje diseñado por Microsoft para simplificar la construcción, distribución y ejecución de aplicaciones para Internet.

La plataforma .NET

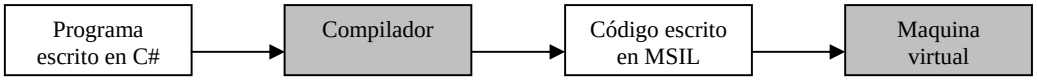
La plataforma consta de las siguientes partes:

- Un conjunto de lenguajes de programación (C#, J#, JScript, C++ gestionado, Visual Básic.NET, y otros proyectos independientes).
- Un conjunto de herramientas de desarrollo (entre ellos Monodevelop o Visual Studio.NET de Microsoft)
- Una librería de clases amplia y común para todos los lenguajes.
- Un sistema de ejecución de Lenguaje Común. (CLR).

- Un conjunto de servidores .NET
- Un conjunto de servicios .NET
- Dispositivos electrónicos con soporte .NET

Uno de los componentes de .NET es la máquina virtual (CLR: *Common Language Runtime*) que procesa código escrito en un lenguaje intermedio (MSIL: *Microsoft Intermediate Language*).

El paquete .NET incluye un compilador de C# que produce un código escrito en un lenguaje intermedio, común para todos los lenguajes de dicha plataforma, que será el que la maquina virtual ejecutará (MSIL).



MSIL es un lenguaje máquina que no es específico de ningún procesador, sino de la máquina virtual de .NET. Se trata de un lenguaje de más alto nivel que otros lenguajes máquina: trata directamente con objetos y tiene instrucciones para cargarlos, guardarlos, iniciarlos, invocar a sus métodos, así como para realizar operaciones aritméticas y lógicas, para controlar el flujo de ejecución, etc.

La máquina virtual posee un recolector de basura (para eliminar los objetos que no estén referenciados) y proporciona traductores del lenguaje intermedio a código nativo para cada arquitectura soportada: compiladores JIT(Just in Time: al instante).

Antes de que el código MSIL pueda ser ejecutado por el procesador de nuestra máquina, debe ser convertido a código nativo. Ésta es la tarea del compilador JIT: producir código nativo para el microprocesador particular de nuestra maquina. El código MSIL es convertido a codigo nativo según se va ejecutando.

C# es independiente de la plataforma (al igual los otros lenguajes del paquete .NET), es decir que el código producido por el compilador C# puede transportarse a cualquier plataforma (Intel, Sparc, motorota, etc.) que tenga instalado una máquina virtual de .NET y ejecutarse.

3.3.3 El lenguaje C#

- C#, pronunciado C Sharp (C bien definido), es actualmente, junto con Java, uno de los lenguajes de programación más populares en Internet.
- C# está disponible para:
    - el desarrollo de aplicaciones de propósito general,
    - aplicaciones que muestren interfaces gráficas,
    - aplicaciones para Internet,
    - aplicaciones para móviles.

- C# es un lenguaje moderno orientado a objetos, simple y potente que permite desarrollar una amplia gama de aplicaciones para la nueva plataforma Microsoft .NET.
- C# permite abordar el desarrollo de aplicaciones complejas con facilidad y rapidez sin sacrificar la potencia y el control que ofrecen C y C++.
- C# es un lenguaje orientado a objetos seguro y elegante que permite a los desarrolladores construir un amplio rango de aplicaciones seguras y robustas que se ejecutan sobre .NET Framework.
- C# se puede utilizar para crear aplicaciones cliente Windows tradicionales, Servicios Web XML, componentes distribuidos, aplicaciones cliente servidor, aplicaciones para bases de datos y muchas otras.
- La palabra "NET" hace referencia al ámbito donde operarán nuestras aplicaciones.
- C# proporciona la tecnología necesaria para saltar desde el desarrollo de aplicaciones cliente-servidor tradicionales a la siguiente generación de aplicaciones escalables para la Web, introduciendo nuevos conceptos, como ensambladores, formularios Web, ADO.NET y el .NET Framework.
- En resumen C# permite escribir tanto programas convencionales como para Internet, permite trabajar con todo tipo de datos, crear estructuras dinámicas, trabajar con ficheros, diseñar interfaces gráficas de usuario, etc.
- C# es una evolución de C/C++.

3.3.4 Instalación del Entorno de Desarrollo

Para poder empezar, lo que un usuario de C# necesita y debe de hacer para desarrollar programas es editar el programa, compilarlo, ejecutarlo y depurarlo, pero para escribir programas se necesita un **entorno de desarrollo C#**.

Una vez que se haya elegido el entorno de desarrollo, lo primero que hay que hacer una vez instalado es asegurarse de que las rutas donde se localizan las herramientas, las bibliotecas, la documentación y los ficheros fuente hayan sido establecidas.
El trabajo de edición, compilación, ejecución y depuración en la plataforma Windows se puede realizar a través de una interfaz de línea de órdenes de SDK o utilizar un entorno de desarrollo con interfaz gráfica de usuario.

Para la compilación y ejecución de los ejemplos y prácticas desarrolladas en este trabajo se utiliza el Kit de desarrollo de software (SDK) de Microsoft .NET Framework 2.0 (x86) y SharpDevelop 2.2, un entorno integrado de desarrollo (IDE) de libre distribución para proyectos C# y VB.NET en la plataforma .NET de Microsoft.

Antes de instalar SharpDevelop, se necesita tener instalado .NET Framework 2.0.
¿Dónde se consigue el paquete .NET?
A través de Internet: http://www.microsoft.com/downloads.

Para instalar el entorno de desarrollo C#,

1) ejecutar el fichero *setup.exe*
2) Se añadirá un menú *Inicio-Programas-Microsoft .NET Framework SDK*

También instalar el entorno de desarrollo SharpDevelop 2.2, ejecutar el fichero SharpDevelop_2.2.1.2648_Setup.

3.4 UNIDAD II: INTRODUCCION A LA PROGRAMACION ORIENTADA A OBJETOS

Título del tema:

Introducción a la Programación Orientada a Objetos

Objetivo:

- Conocer los conceptos, las características y algunas ventajas de la Programación Orientada a Objetos.

Contenido:

- Programación Orientada a Objetos
- Mecanismos básicos de la POO
- Características de la POO
- Lenguajes Orientados a Objetos
- Ventajas de los lenguajes Orientados a Objetos

Duración:

- 3 horas

Bibliografía básica:

- http://www.monografias.com/trabajos14/progorie/progorie.shtml
- http://www.lenguajes-de-programacion.com/programacion-orientada-a-objetos.shtml

3.4 UNIDAD II: INTRODUCCION A LA PROGRAMACION ORIENTADA A OBJETOS

3.4.1 Programación Orientada a Objetos (POO)

Se puede definir **POO** como una técnica o estilo de programación que utiliza **objetos**, el cual es una entidad que contiene en sí misma los datos (valores de sus atributos) que la identifican y los algoritmos (métodos) que permiten manipular dichos datos. Pensar en términos de objetos es muy parecido a cómo lo haríamos en la vida real.

La Programación Orientada a Objetos (POO) es un paradigma de la programación de computadores. Cuando se habla de paradigma de programación se hace referencia al conjunto de teorías, estándares, modelos y métodos que permiten organizar el conocimiento, proporcionando un medio bien definido para visualizar el dominio e implementar en un lenguaje de programación la solución de un tipo de problema.
Los diferentes tipos de paradigmas de programación, requieren de una manera propia de pensar y enfocar el problema a resolver, pues cada uno de ellos tiene un marco de referencia conceptual que permite modelar el dominio del problema.

3.4.2 Mecanismos básicos de la POO

La programación orientada a objetos introduce nuevos conceptos, entre ellos destacan los siguientes:

- **Clase**: definiciones de las propiedades y comportamiento de un tipo de objeto concreto. La instanciación es la lectura de estas definiciones y la creación de un objeto a partir de ellas, (de c a d),Es la facilidad mediante la cual la clase D ha definido en ella cada uno de los atributos y operaciones de C, como si eso atributos y operaciones hubiesen sido definidos por la misma D.
- **Objeto**: entidad provista de un conjunto de propiedades o atributos (datos) y de comportamiento o funcionalidad (métodos). Corresponden a los objetos reales del mundo que nos rodea, o a objetos internos del sistema (del programa). Es una instancia a una clase.
- **Método**: algoritmo asociado a un objeto (o a una clase de objetos), cuya ejecución se desencadena tras la recepción de un "mensaje". Desde el punto de vista del comportamiento, es lo que el objeto puede hacer. Un método puede producir un cambio en las propiedades del objeto, o la generación de un "evento" con un nuevo mensaje para otro objeto del sistema.
- **Evento**: un suceso en el sistema (tal como una interacción del usuario con la máquina, o un mensaje enviado por un objeto). El sistema maneja el evento enviando el mensaje adecuado al objeto pertinente. También se puede definir como evento, a la reacción que puede desencadenar un objeto, es decir la acción que genera.
- **Mensaje**: una comunicación dirigida a un objeto, que le ordena que ejecute uno de sus métodos con ciertos parámetros asociados al evento que lo generó.

- **Propiedad o atributo**: contenedor de un tipo de datos asociados a un objeto (o a una clase de objetos), que hace los datos visibles desde fuera del objeto y esto se define como sus características predeterminadas, y cuyo valor puede ser alterado por la ejecución de algún método.
- **Estado interno**: es una propiedad invisible de los objetos, que puede ser únicamente accedida y alterada por un método del objeto, y que se utiliza para indicar distintas situaciones posibles para el objeto (o clase de objetos).
- **Componentes de un objeto**: atributos, identidad, relaciones y métodos.
- **Representación de un objeto**: un objeto se representa por medio de una tabla o entidad que esté compuesta por sus atributos y funciones correspondientes.

3.4.3 Características de la POO

Hay un cierto desacuerdo sobre exactamente qué características de un método de programación o lenguaje le definen como "orientado a objetos", pero hay un consenso general en que las características siguientes son las más importantes:

- **Abstracción**: Cada objeto en el sistema sirve como modelo de un "agente" abstracto que puede realizar trabajo, informar y cambiar su estado, y "comunicarse" con otros objetos en el sistema sin revelar *cómo* se implementan estas características. Los procesos, las funciones o los métodos pueden también ser abstraídos y cuando lo están, una variedad de técnicas son requeridas para ampliar una abstracción.
- **Encapsulamiento**: Significa reunir a todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. Esto permite aumentar la cohesión de los componentes del sistema. Algunos autores confunden este concepto con el principio de ocultación, principalmente porque se suelen emplear conjuntamente.
- **Principio de ocultación**: Cada objeto está aislado del exterior, es un módulo natural, y cada tipo de objeto expone una *interfaz* a otros objetos que especifica cómo pueden interactuar con los objetos de la clase. El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas, solamente los propios métodos internos del objeto pueden acceder a su estado. Esto asegura que otros objetos no pueden cambiar el estado interno de un objeto de maneras inesperadas, eliminando efectos secundarios e interacciones inesperadas. Algunos lenguajes relajan esto, permitiendo un acceso directo a los datos internos del objeto de una manera controlada y limitando el grado de abstracción. La aplicación entera se reduce a un agregado o rompecabezas de objetos.
- **Polimorfismo**: comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre, al llamarlos por ese nombre se utilizará el comportamiento correspondiente al objeto que se esté usando. O dicho de otro modo, las referencias y las colecciones de objetos pueden contener objetos de diferentes tipos, y la invocación de un comportamiento en una referencia producirá el comportamiento correcto para el tipo real del objeto referenciado. Cuando esto ocurre en "tiempo de ejecución", esta última característica se llama

asignación tardía o *asignación dinámica*. Algunos lenguajes proporcionan medios más estáticos (en "tiempo de compilación") de polimorfismo, tales como las plantillas y la sobrecarga de operadores de C++.

- **Herencia**: las clases no están aisladas, sino que se relacionan entre sí, formando una jerarquía de clasificación. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen. La herencia organiza y facilita el polimorfismo y el encapsulamiento permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. Estos pueden compartir (y extender) su comportamiento sin tener que reimplementar su comportamiento. Esto suele hacerse habitualmente agrupando los objetos en *clases* y estas en *árboles* o *enrejados* que reflejan un comportamiento común. Cuando un objeto hereda de más de una clase se dice que hay *herencia múltiple*.

3.4.4 Lenguajes orientados a objetos

Hay un gran cantidad de lenguajes orientados a objetos, entre los lenguajes orientados a objetos destacan algunos como:

C++ , C# , Visual Basic, Clarion , Python, Lenguaje de programación D , Object Pascal (Delphi) , Harbour , Eiffel , Java , JavaScript (la herencia se realiza por medio de la programación basada en prototipos) entre otros.

Un nuevo paso en la abstracción de paradigmas de programación es la Programación Orientada a Aspectos (POA). Aunque es todavía una metodología en estado de maduración, cada vez atrae a más investigadores e incluso proyectos comerciales en todo el mundo.

3.4.5 Ventajas de los lenguajes orientados a objetos

- Fomenta la reutilización y extensión del código.
- Permite crear sistemas más complejos.
- Relacionar el sistema al mundo real.
- Facilita la creación de programas visuales.
- Construcción de prototipos
- Agiliza el desarrollo de software
- Facilita el trabajo en equipo
- Facilita el mantenimiento del software

Lo interesante de la POO es que proporciona conceptos y herramientas con las cuales se modela y representa el mundo real tan fielmente como sea posible.

3.5 UNIDAD III: CLASES

Titulo del tema:

Clases

Objetivos:

- Entender el concepto de clase y objeto.
- Aplicar los conceptos de abstracción en la implementación de una clase.

Contenido:

- Definición de una clase
- Miembros de una clase
- Control de acceso a los miembros de la clase
- Implementación de una clase: CFecha
- Métodos sobrecargados
- Referencia this
- Construcción de objetos
- Asignación de objetos
- Constructor copia
- Destrucción de objetos

Duración:

- 6 horas

Bibliografía básica:

- Ceballos Sierra, Francisco Javier. Microsoft C# Curso de Programación. Editorial RAMA.

3.5 UNIDAD III: CLASES

3.5.1 Definición de una clase

Una clase es un tipo definido por el usuario que contiene básicamente:

- **Atributos**: definen el estado de un determinado objeto (denominados también *campos*).
- **Propiedades**: representan información sobre un objeto.
- **Métodos**: representan acciones que un objeto puede realizar.

Los atributos y los métodos se denominan en general *miembros* de la clase.

Para definir una clase, debemos seguir la sintaxis siguiente:

```
class nombre_clase
{
    //cuerpo de la clase: atributos y metodos
}
```

El cuerpo de la clase en general consta de modificadores de acceso (**public, protected y private**), atributos, propiedades, mensajes y métodos).

Ejemplo:

```
class Esfera
{
    //Miembros privados
    private double radio;

    //Miembros protegidos
    protected void MsgEsNegativo()
    {
        Console.WriteLine("El radio es negativo. Se convierte a
positivo.");
    }

    //Miembros publicos
    public Esfera()
    {
    }//Constructor sin parametros

    public Esfera(double r)
    {
        if (r < 0)
        {
            MsgEsNegativo();
            r = -r;
        }
        radio = r;
    }
```

```
    public double VolumenEsfera()
    {
        return (4.0 / 3.0) * Math.PI * Math.Pow(radio, 3);
    }

    public double AreaEsfera()
    {
        return 4 * Math.PI * radio * radio;
    }

    public double Radio
    {
        get
        {
            return radio;
        }
        set
        {
            if (value < 0)
            {
                MsgEsNegativo();
                value = -value;
            }
            radio = value;
        }
    }
}
```

La clase *Esfera* define un nuevo tipo de datos y podemos declarar objetos de tipo *Esfera* como cualquier otra variable:

```
Esfera e1, e2, e3;
```

Los objetos e1, e2 y e3 por ser de tipo *Esfera*, cada uno de ellos tiene su propia copia de los atributos *x*, *y* y radio, pero no sucede lo mismo con los métodos MsgEsNegativo, VolumenEsfera, AreaEsfera, de las cuales sólo existe una copia que es compartida por todos los objetos.

3.5.2 Miembros de una clase

Los miembros de una clase tienen las siguientes características:

Atributos

- Los atributos constituyen la estructura interna de los objetos de una clase.
- Cada atributo de una clase debe tener un nombre único.
- Se puede asignar un valor inicial a un atributo de una clase, aunque esto no se hace, porque es una operación típica del constructor de la clase.
- Referencias a otros objetos de clases existentes, también podemos declararlos como atributos de una clase.

Métodos

- Los métodos forman lo que se denomina *interfaz* o medio de acceso a la estructura interna de los objetos.
- Los métodos definen las operaciones que se pueden realizar con sus atributos.
- El conjunto de todos los métodos se corresponde con el conjunto de mensajes a los que los objetos de una clase puede responder.
- En C#, un método siempre pertenece a una clase y su definición nunca puede contener a la definición de otro método; esto es, C# no permite métodos anidados.

Propiedades

- Una propiedad es un miembro, normalmente público, de una clase que proporciona un mecanismo para asignar u obtener el valor de un atributo privado de la clase.
- Es un método con uno o dos *descriptores de acceso* (**get**, **set**), con un parámetro implícito, **value**, que recibe el valor a asignar, y que retorna el valor del atributo al que se refiere.
- Estos descriptores de acceso son rutinas de código declaradas dentro de la propiedad para recuperar (**get**) o establecer (**set**) el valor de la misma.

Su sintaxis es la siguiente:

```
public tipo nombre_propiedad
{
        get
        {
           // Aquí se devuelve “return” el valor del atributo
        }
        set
        {
           //Aquí se asigna el valor “value” al atributo
        }
}
```

- Normalmente los descriptores de acceso de una propiedad se definen por parejas, utilizando **get** y **set**.
- Ahora bien, si la propiedad es de sólo lectura (sólo **get**) o de sólo escritura (sólo **set**) puede definirse cada descriptor de acceso de forma individual.

3.5.3 Control de acceso a los miembros de la clase

Básicamente no se puede acceder directamente a los atributos de un objeto, sino que hay que hacerlo a través de las propiedades y métodos de la clase, esto es la ocultación de datos. Es decir, que el usuario de la clase sólo tendrá acceso a una o más propiedades y/o métodos que le permitirán acceder a los miembros privados, ignorando la disposición de éstos.
Para controlar el acceso a los miembros de una clase, C# provee las palabras clave, denominadas *modificadores de acceso*, **private** (privado), **protected** (protegido) y **public** (público), se pueden omitir, en cuyo caso se consideran de acceso privado. Otros modificadores son **internal** y **protected internal**.
A continuación veremos el significado de cada una de estas palabras clave:

- **Acceso público**

Un miembro de una clase declarado **public** (público) puede ser accedido por un objeto de esa clase en cualquier parte donde el objeto en cuestión sea accesible. No hay restricciones de acceso.

- **Acceso privado**

Un miembro de una clase declarado **private** (privado) puede ser accedido por un objeto de esa clase sólo desde los métodos y propiedades de dicha clase. Esto significa que no puede ser accedido por los métodos y propiedades de cualquier otra clase, incluyendo las subclases.

- **Acceso protegido**

Un miembro de una clase declarado **protected** (protegido) se comporta exactamente igual que uno privado para los métodos y propiedades de cualquier otra clase, excepto para los métodos y propiedades de sus subclases, para las que se comporta como miembro público.

- **Acceso interno**

Un miembro de una clase declarado **internal** (interno) puede ser accedido por un objeto de esa clase en cualquier parte de la aplicación actual (ensamblado actual) donde el objeto en cuestión sea accesible.

Un miembro de una clase sólo puede ser accedido, implícita o explícitamente, por un objeto de esa clase. Por ejemplo:

```
public class Test
{
     public static void Main(string[] args)
     {
          Esfera e = new Esfera(6.8);
          double area = e.AreaEsfera();
          double vol = e.VolumenEsfera();
     }
}
```

3.5.4 Implementación de una clase: CFecha

Para ejemplificar todos los conceptos expuestos hasta ahora, se lista a continuación un programa que hace uso de una clase CFecha:

```
    public class CFecha
    {
        //Atributos
        private int dia, mes, anyo;
```

```
        //Metodos
        protected bool Bisiesto()
        {
            return ((anyo % 4 == 0) && (anyo % 100 != 0) || (anyo % 400
== 0));
        }

        public void AsignarFecha(int dd, int mm, int aaaa)
        {
            dia = dd;
            mes = mm;
            anyo = aaaa;
        }

        public void ObtenerFecha(out int dd, out int mm, out int aaaa)
        {
            dd = dia;
            mm = mes;
            aaaa = anyo;
        }

        public bool FechaCorrecta()
        {
            bool diaCorrecto, mesCorrecto, anyoCorrecto;
            //anyo correcto?
            anyoCorrecto = (anyo >= 1582);

            //mes correcto?
            mesCorrecto = (mes >= 1) && (mes <= 12);

            switch (mes)
            //dia correcto?
            {
                case 2:
                    if (Bisiesto())
                        diaCorrecto = (dia >= 1 && dia <= 29);
                    else
                        diaCorrecto = (dia >= 1 && dia <= 28);
                    break;
                case 4: case 6:case 9: case 11:
                    diaCorrecto = (dia >= 1 && dia <= 30);
                    break;
                default:
                    diaCorrecto = (dia >= 1 && dia <= 31);
                    break;
            }
            return diaCorrecto && mesCorrecto && anyoCorrecto;
        }

        //Visualizar una fecha
        public void VisualizarFecha(CFecha fecha)
        {
            int d, m, a;
            fecha.ObtenerFecha(out d, out m, out a);
            Console.WriteLine(d + "/" + m + "/" + a);
        }
```

```
        //Leer una fecha
        public void LeerFecha(ref int d, ref int m, ref int a)
        {
            Console.WriteLine("dia, ##    : ");
            d = Leer.datoInt();
            Console.WriteLine("mes, ##    : ");
            m = Leer.datoInt();
            Console.WriteLine("anyo, ##: ");
            a = Leer.datoInt();
        }
    }
```

Para comprobar que la clase CFecha que acabamos de diseñar trabaja correctamente, podemos escribir una aplicación Test según se muestra a continuación:

```
public class Test
{
     //Establecer una fecha, verificarla y visualizarla
     public static void Main(string[] args)
     {
         CFecha fecha1 = new CFecha();
         int dia = 1 , mes = 1, anyo = 2001;
         do
         {
             fecha1.LeerFecha(ref dia, ref mes, ref anyo);
             fecha1.AsignarFecha(dia, mes, anyo);
         } while (!fecha1.FechaCorrecta());
         fecha1.VisualizarFecha(fecha1);
         Console.ReadLine();
     }
}
```

Al compilar las clases que contiene nuestra aplicación, el compilador C# generará un fichero *.exe* denominado *ensamblado*.
En la clase CFecha los atributos declarados privados y los métodos protegidos sólo son accesibles por los métodos de su clase, es decir, que si un método de otra clase intenta acceder a uno de estos atributos, el compilador genera un error.
Todos los métodos declarados públicos son accesibles, además de por los métodos de su clase, por cualquier otro método de otra clase.

3.5.5 Métodos sobrecargados

Cuando en una clase un mismo método se define varias veces con distinto número de parámetros, o bien con el mismo número de parámetros pero diferenciándose una definición de otra en que al menos un parámetro es de un tipo diferente, se dice que el método está *sobrecargado*.

Los métodos sobrecargados pueden diferir también en el tipo de valor retornado.
El compilador C# no admite que se declaren dos métodos que sólo difieran en el tipo del valor retornado; deben diferir también en la lista de parámetros; lo que importa es el número y el tipo de parámetros.

La sobrecarga de métodos elimina la necesidad de definir métodos diferentes que en esencia hacen lo mismo, como es el caso del método **WriteLine**, o también hace posible que un método se comporte de una u otra forma según el número de argumentos con el que sea invocado.

Cuando una clase sobrecarga un método una o más veces, ¿cómo sabe C# cuál tiene que ejecutar cuando se invoque? Pues esto lo hace comparando el número y los tipos de argumentos especificados en la llamada, con los parámetros especificados en las distintas definiciones del método. Por ejemplo, a continuación se muestra las diferentes formas de invocar al método *AsignarFecha*:

```
fecha1.AsignarFecha();
fecha1.AsignarFecha(dia);
fecha1.AsignarFecha(dia, mes);
fecha1.AsignarFecha(dia, mes, anyo);
```

Si el compilador C# no encuentra un método exactamente con los mismos tipos de argumentos especificados en la llamada, realizaría sobre dichos argumentos las conversiones implícitas permitidas entre tipos, tratando de adaptarlos a alguna de las definiciones existentes del método. Si este intento fracasa, entonces se producirá un error.

C# soporta métodos con un número variable de parámetros, todos del mismo tipo. Esta característica se especifica anteponiendo la palabra reservada **params** a la definición del parámetro del método, el cual debe ser una matriz unidimensional.

Por ejemplo, todas las sobrecargas del método *AsignarFecha* podrían ser sustituidas por el código siguiente y cualquiera de las llamadas anteriores seria correcta:

```
public void AsignarFecha(params int[] fecha)
{
    //Obtener la fecha actual
    int dd, mm, aaaa;
    DateTime fechaActual = DateTime.Now;
    dd = fechaActual.Day;
    mm = fechaActual.Month;
    aaaa = fechaActual.Year;

    //Asignar la fecha pasada (por omision la fecha actual)
    if (0 < fecha.Length)
         dia = fecha[0];
    else
         dia = dd;
    if (1 < fecha.Length)
         mes = fecha[1];
    else
         mes = mm;
    if (2 < fecha.Length)
         anyo = fecha[2];
    else
         anyo = aaaa;
```

```
}
```

3.5.6 Referencia this

Cada objeto mantiene su propia copia de los atributos pero no de los métodos de su clase, de los cuales solo existe una copia para todos los objetos de esa clase. Es decir, que cada objeto almacena sus propios datos, pero para acceder y operar con ellos, todos comparten los mismos métodos definidos en su clase. Por lo tanto, para que un método conozca la identidad del objeto particular para el que ha sido invocado, C# proporciona una referencia al objeto denominada **this**.

Cuando creamos un objeto y le enviamos un mensaje, C# define la referencia **this** de soloo lectura para referirse al objeto creado, en el cuerpo del método que se ejecuta como respuesta al mensaje. Esa definicion seria equivalente a la siguiente:

```
readonly nombreClase this = objeto;
```

Ejemplo:

```
public void AsignarFecha(int dd, int mm, int aaaa)
{
      this.dia = dd;
      this.mes = mm;
      this.anyo = aaaa;
}
```

¿Qué representa **this** en este método?
this es una referencia al objeto que recibió el mensaje AsignarFecha, esto es, al objeto sobre el que se está realizando el proceso llevado a cabo por el método AsignarFecha.

3.5.7 Construcción de objetos

- Un constructor es un método especial de una clase que es llamado automáticamente siempre que se crea un objeto de esa clase. Su función es iniciar el objeto.
- Un constructor tiene el mismo nombre que la clase a la que pertenece y no puede retornar un valor (ni siquiera **void**), ni se puede heredar.
- Puede ser declarado como **public, private, protected, internal o protected internal**.
- Cuando no se escribe explícitamente un constructor, C# asume uno por omisión.
  Así:

```
public CFecha()
{
}
```

- Un constructor por omisión de una clase es un constructor sin parámetros que no hace nada.

- Pero es necesario porque será invocado cada vez que se construya un objeto sin especificar ningún argumento, en cuyo caso el objeto será iniciado por los valores predeterminados por el sistema (los atributos numéricos a ceros, los alfabéticos a nulos, y las referencias a objetos a **null**).
- Siempre que en una clase se define explícitamente un constructor, el constructor implícito (constructor por omisión) es reemplazado por éste. Por eso se tiene que definir explícitamente, de lo contrario, intentar crear un objeto sin especificar parámetros daría lugar a un error.
- Los constructores, salvo en casos excepcionales, deben declararse siempre públicos para que puedan ser invocados desde cualquier parte de una aplicación donde se cree un objeto de su clase.
- Cuando el constructor tenga parámetros, para crear un nuevo objeto hay que especificar la lista de argumentos correspondientes entre los paréntesis que siguen al nombre de la clase del objeto.

Ejemplo:

```
public static void Main(string[] args)
{
   CFecha fecha1 = new CFecha(16, 6, 2022); //Linea invoca al
constructor con parámetros de la clase CFecha

}
```

```
  //Constructor con parámetros
  public CFecha(int dd, int mm, int aaaa)
  {
        dia = dd;
        mes = mm;
        anyo = aaaa;
 }
```

- Al igual que como se sobrecarga un método, se puede sobrecargar el constructor de una clase, y se puede invocar al constructor con 0, 1, 2 ó 3 argumentos.
- A diferencia de los otros métodos de la clase, un constructor no puede ser invocado explícitamente, pero si implícitamente a través de **this**.
- Para llamar a un constructor en la clase actual desde otro constructor utilizaremos la sintaxis siguiente:

```
public nombre_constructor(argumentos) : this(argumentos)
{
}
```

3.5.8 Asignación de objetos

- Cuando trabajamos con objetos lo que realmente manipulamos desde cualquier método son referencias a los objetos.

```
CFecha ayer = new CFecha();
```

```
CFecha hoy = new CFecha();
CFecha mañana = new CFecha();
CFecha otrodia = ayer; //declara una referencia otrodia y le asigna
ayer; otrodia y ayer son referencias a objetos CFecha, que apuntan al
mismo objeto ayer
mañana = hoy; //asigna hoy a mañana, ambas referencias apuntan al
objeto referenciado por hoy
```

- El operador de asignación no sirve par copiar un objeto en otro. Tenemos que añadir a la clase `CFecha` un método como el siguiente:

```
public void Copiar(CFecha objFecha)
{
     dia = objFecha.dia;
     mes = objFecha.mes;
     anyo = objFecha.anyo;
}
```

- El método `Copiar` copia miembro a miembro el objeto pasado como argumento en el objeto que recibe el mensaje `Copiar`.

Ejemplo:

```
ayer.Copiar(hoy); //Copia el objeto hoy en el objeto ayer
```

- Tenemos que modificar el método Copiar de la siguiente forma para poder realizar copias múltiples encadenadas.

Ejemplo:

```
public CFecha Copiar(CFecha objFecha)
{
     dia = objFecha.dia;
     mes = objFecha.mes;
     anyo = objFecha.anyo;
     return this;
}
```

3.5.9 Constructor copia

Se puede iniciar un objeto asignándole otro objeto de su misma clase en el momento de su creación. Un constructor copia es aquel que se invoca para iniciar un nuevo objeto creado a partir de otro existente.

La sintaxis de este constructor es la siguiente:

```
modificadores nombre_clase(nombre_clase referencia_objeto)
```

Ejemplo:

```
//Constructor copia
public CFecha(CFecha objFecha)
{
      dia = objFecha.dia;
      mes = objFecha.mes;
      anyo = objFecha.anyo;
}
```

```
public static void Main(string[] args)
{
    CFecha ayer = new CFecha(1,3, 2010);
    CFecha hoy = new CFecha(ayer); //llama al constructor copia
    //...
}
```

3.5.10 Destrucción de objetos

El destructor es un método especial de una clase que se ejecuta antes de que un objeto de esa clase sea eliminado físicamente de la memoria. Tiene las siguientes características:

- Una clase sólo puede tener un destructor.
- Tiene el mismo nombre que la clase a la que pertenece precedido por ~.
- Un destructor no permite modificaciones de acceso ni tiene parámetros.
- Se invoca automáticamente justo antes de que el objeto sea recolectado como basura por el recolector de basura de C#.
- No se puede llamar explícitamente.
- Un destructor no se puede heredar ni sobrecargar.
- El compilador C# crea un destructor por omisión si en la clase no lo especificamos.

La sintaxis de un destructor es la siguiente:

```
~nombre_clase()
{ /* sin codigo */ }
```

Ejemplo:

```
//Destructor
~CFecha()
{
     Console.WriteLine("Objeto destruido");
}
```

El destructor llama explícitamente al método **Finalize** de la clase **Object** en la clase base del objeto:

```
protected override void Finalize()
{
    try
    {
    }
    finally
```

```
    {
        base.Finalize();
    }
}
```

3.6 UNIDAD IV: OPERADORES SOBRECARGADOS

Título del tema:

Operadores sobrecargados

Objetivo:

- Entender el concepto de operadores sobrecargados.
- Mostrar el uso de sobrecargar un operador en la implementación de una clase.
- Aprender a sobrecargar operadores de asignación, aritméticos, binarios y unarios.

Contenido:

- Sobrecargar un operador
- Ejemplo: la clase CComplejo
- Sobrecarga de operadores binarios
- Sobrecarga de operadores de asignación y operadores aritméticos
- Aritmética mixta
- Sobrecarga de operadores unarios
- Operadores unarios/binarios
- Conversiones personalizadas
- Indexación

Duración:

- 6 horas

Bibliografía básica:

- Ceballos Sierra, Francisco Javier. Microsoft C# Curso de Programación. Editorial RAMA.

3.6 UNIDAD IV: OPERADORES SOBRECARGADOS

3.6.1 Sobrecargar un operador

La sobrecarga de un operador consiste en un operador que es capaz de desarrollar su función en varios contextos diferentes sin necesidad de otras operaciones adicionales.

C# provee la facilidad de asociar un método a un operador estándar, con el fin de que el método sea llamado cuando el compilador detecte este operador en un contexto específico. Se dice entonces que el *operador* está *sobrecargado*, porque los conjuntos de objetos sobre los que puede operar son más.

La sintaxis para declarar un operador sobrecargado es la siguiente:

```
public static tipo operator operador(parametros)
{
      //cuerpo
}
```

No todos los operadores se pueden sobrecargar y algunos presentan restricciones, como se indica en la siguiente tabla:

Operadores	Posibilidad de sobrecarga
+, -, !, ~, ++, --, true, false	Estos operadores unarios sí se pueden sobrecargar.
+, -, *, /, %, &, |, ^, <<, >>	Estos operadores binarios sí se pueden sobrecargar.
==, !=, <, >, <=, >=	Los operadores de comparación se pueden sobrecargar, dicha sobrecarga debe realizarse en parejas; es decir, si se sobrecarga **==**, también se debe sobrecargar **!=**. Lo contrario también es cierto y similar para < y >, y para <= y >=.
&&, ||	Los operadores lógicos condicionales no se pueden sobrecargar, pero se evalúan mediante **&** y |, los cuales sí se pueden sobrecargar.
[ ]	El operador de indexación de matrices no se puede sobrecargar, pero se pueden definir indizadores.
( )	El operador de conversión explícita de tipos no se puede sobrecargar, pero se pueden definir nuevos operadores de conversión: **explicit** e **implicit**.
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Los operadores de asignación no se pueden sobrecargar, pero +=, por ejemplo, se evalúa con +, el cual sí se puede sobrecargar.

=, ., ?:, ->, new, is, sizeof, typeof	Estos operadores no se pueden sobrecargar.

Un operador puede ser:

- Binario: si se aplica sobre dos operandos.
- Unario: si se aplica sobre un único operando.

La sobrecarga de operadores es usada con clases y con estructuras.

Ejemplo:

```
public class ejemOpe
{
    //Atributos
    public static ejemOpe operator +(ejemOpe a)
    {
        //...
    }
    public static ejemOpe operator +(ejemOpe a, ejemOpe b)
    {
        //...
    }
};

class Test
{
    public static void Main()
    {
        //x, y, y z son objetos de la clase ejemOpe
        ejemOpe x = new ejemOpe(), y = new ejemOpe(), z = new
ejemOpe();
        y = -z;     //menos unario
        z = x - y;  //menos binario
    }
}
```

En el código anterior, la sentencia y = -z implícitamente invoca al método *operator* + con un parámetro y z = x – y, al método *operator* – con dos parámetros.

```
public struct ejemOpe
{
    //Atributos
    public static ejemOpe operator +(ejemOpe a)
    {
        //...
    }
    public static ejemOpe operator +(ejemOpe a, ejemOpe b)
    {
        //...
    }
```

```
};


class Test
{
    public static void Main()
    {
        //x, y, y z son objetos de la clase ejemOpe
        ejemOpe x = new ejemOpe(), y, z = new ejemOpe();
        y = -z;     //menos unario
        z = x - y;  //menos binario
    }
}
```

Una estructura se trata de un tipo valor por lo que se pueden crear objetos de un tipo estructura sin utilizar un operador **new**.

Cuando se sobrecarga un operador, esté conserva su propiedad de binario o unario, y mantiene invariable su prioridad de evaluación y su asociatividad.

Se recomienda hacer sobrecargas que no realicen una operación diferente a la esperada por el operador utilizado.

Los operadores sobrecargados son útiles cuando se trata de trabajar con tipos abstractos de datos que definen objetos pertenecientes al campo de las matemáticas. Por ejemplo, operaciones con números complejos.

3.6.2 Ejemplo: la clase CComplejo

El ejemplo de los complejos, desde el punto de vista de un diseño orientado a objetos, haciendo uso de estructuras, es el siguiente:

```
//Complejo.cs - Declaracion de la estructura complejo
using System;

namespace OperSobre
{
    public struct Complejo
    {
        private double real; //parte real
        private double imag; //parte imaginaria

        public Complejo(double r, double i)
        {
            real = r; imag = i;
        }

        //Asignacion de Complejos
        public void AsignarComplejo(double r, double i)
        {
            real = r; imag = i;
        }
```

```
        //suma de complejos
        public static Complejo operator +(Complejo x, Complejo y)
        {
            return new Complejo(x.real + y.real, x.imag + y.imag);
        }

        //Diferencia de Complejos
        public static Complejo operator -(Complejo x, Complejo y)
        {
            return new Complejo(x.real - y.real, x.imag - y.imag);
        }

        //Mostrar un numero complejo
        public override string ToString()
        {
            return (String.Format("{0}" + (imag >= 0 ? "+" : " ") +
"{1}i", real, imag));
        }
    }
}
```

```
using System;

namespace OperSobre
{
    class Test
    {
        public static void Main(string[] args)
        {
            Complejo a = new Complejo();
            Complejo b;
            Complejo c = new Complejo(1.5, 2);
            Complejo d;
            double re, im;

            Console.WriteLine("Numero complejo: ");
            Console.WriteLine("re: ");
            re = Leer.datoDouble();
            Console.WriteLine("im: ");
            im = Leer.datoDouble();
            a.AsignarComplejo(re, im);

            b = a;
            d = a + b - c;
            d = d + new Complejo(5, 2);
            Console.WriteLine(d.ToString());

        }
    }
}
```

Restricciones

- No se puede sobrecargar un operador binario para crear un operador unario.

- No se puede sobrecargar un operador unario para realizar operaciones binarias.

- Los operadores que pueden actuar como unarios y binarios, se pueden sobrecargar para utilizarlos en uno u otro contexto.

- Aunque es posible modificar la definición de un operador (^), no es posible modificar su precedencia (prioridad).

3.6.3 Sobrecarga de operadores binarios

A continuación veremos cómo sobrecargar los operadores binarios, específicamente los operadores de asignación y aritméticos, los de relación, y los de E/S.
Dentro de este conjunto de operadores los hay que modifican la estructura interna del objeto, como +=, y operadores que producen un nuevo objeto, como +.

3.6.4 Sobrecarga de operadores de asignación y operadores aritméticos

Los operadores de asignación, como =, +=, -=, *=, etc. no se pueden sobrecargar. No obstante, cuando trabajamos con estructuras si se puede hacer que se realicen las operaciones correspondientes.
Por ejemplo, para mostrar lo anterior se hará uso de la siguiente aplicación, la estructura **CTiempo** se muestra más adelante.

```
public static void Main(string[] args)
{
    CTiempo hora = new CTiempo();
    CTiempo Hoy;
    // . . .
    hora.AsignarTiempo(6, 43, 23);
    Hoy = hora; //el valor de hora es copiado en Hoy
    //. . .
}
```

El operador = no de puede sobrecargar, no obstante, trabajando con estructuras cuando se ejecuta la operación `Hoy = hora,` el valor de `hora` es copiado en `Hoy`.

```
public static void Main(string[] args)
{
     CTiempo hora = new CTiempo();
     CTiempo Hoy;
     CTiempo Ayer = new CTiempo(9, 56, 42);
     CTiempo Pasado;
     hora.AsignarTiempo(6, 43, 23);
     Hoy = hora;
     Pasado = hora + Ayer;
     // . . .
     hora += Pasado; // equivale hora = hora + Pasado
     // . . .
}
```

Para el operador de asignación +=, no se puede sobrecargar; la solución es sobrecargar el operador +, lo que permitirá que primero se realice la suma y después la asignación. Cuando se ejecuta la sentencia `hora += Pasado,` invoca al método **operador +**, y después al operador =, para sumar dos Tiempos y almacenar el resultado en **hora**.

Los operadores aritméticos, como +, producen un nuevo objeto a partir de sus dos operandos. Para definirlos podemos proceder de dos formas: escribiendo los métodos para que realicen las operaciones a partir de los atributos, o bien para que las realicen invocando a otros métodos ya implementados.

Ejemplo:

```
//suma de dos tiempos
public static CTiempo operator +(CTiempo a, CTiempo b)
{
      CTiempo suma = new CTiempo();
      suma = new CTiempo(a.horas + b.horas, a.minutos + b.minutos,
a.segundos + b.segundos);
      if (suma.segundos > 60)
      {
          suma.segundos -= 60;
          suma.minutos++;
      }
      if (suma.minutos > 60)
      {
          suma.minutos -= 60;
          suma.horas++;
      }
          return suma;
}
```

En el ejemplo anterior se ha sobrecargado el operador + para permitir la suma de dos objetos de tipo **CTiempo**. Observe que después de realizar la suma, se comprueba si los segundos son mayores que 60, es este caso, se le resta 60 y se incrementa en uno a los minutos y si los minutos son mayores que 60, se le resta 60 y se incrementa en uno a la hora para que el resultado sea una hora correcta. En esta versión crea un objeto local suma invocando al constructor **CTiempo** con los valores resultantes de la suma, y devuelve suma como resultado.

Aplicaremos la sobrecarga de estos operadores a una estructura **CTiempo.**

```
using System;

namespace CTiempo
{
    public struct CTiempo
    {
        //Atrtibutos
        private int horas;
        private int minutos;
        private int segundos;
```

```
        //metodos

        public CTiempo(int h, int m, int s)
        {
            horas = h;
            minutos = m;
            segundos = s;
        }

        //Asignacion de Tiempo
        public void AsignarTiempo(int h, int m, int s)
        {
            horas = h;
            minutos = m;
            segundos = s;
        }

        //suma de dos tiempos
        public static CTiempo operator +(CTiempo a, CTiempo b)
        {
            CTiempo suma = new CTiempo();
            suma = new CTiempo(a.horas + b.horas, a.minutos +
b.minutos, a.segundos + b.segundos);
            if (suma.segundos >= 60)
            {
                suma.segundos -= 60;
                suma.minutos++;
            }
            if (suma.minutos >= 60)
            {
                suma.minutos -= 60;
                suma.horas++;
            }
            return suma;
        }

        //Mostrar Tiempo
        public override string ToString()
        {
            return (String.Format("{0}" + ":" + "{1}" + ":" + "{2}",
horas, minutos, segundos));
        }
    };
}
```

```
using System;

namespace CTiempo
{
    public class Test
    {
        public static void Main(string[] args)
        {
            CTiempo hora = new CTiempo();
            CTiempo Hoy;
            CTiempo Ayer = new CTiempo(9, 56, 42);
```

```
            CTiempo Pasado;
            hora.AsignarTiempo(6, 43, 23);
            Hoy = hora;
            Pasado = hora + Ayer;
            Console.WriteLine("La hora de hoy es ...");
            Console.WriteLine(Hoy.ToString());
            Console.WriteLine("La hora de ayer es ...");
            Console.WriteLine(Ayer.ToString());
            Console.WriteLine("Sumando estas dos horas resulta ...");
            Console.WriteLine(Pasado.ToString());
            hora += Pasado;
            Console.WriteLine("la hora es ...");
            Console.WriteLine(hora.ToString());
        }
    }
}
```

Al ejecutar el programa anterior los resultados son los siguientes:

```
La hora de hoy es ...
6:43:23
La hora de ayer es ...
9:56:42
Sumando estas dos horas resulta ...
16:40:5
la hora es ...
23:23:28
```

3.6.5 Aritmética mixta

La aritmética mixta se refiere a la realización de operaciones con operandos no necesariamente del mismo tipo. Por ejemplo, en el siguiente código se suma un objeto de tipo **CTiempo** con un valor de tipo entero:

```
CTiempo a = new CTiempo(4, 59, 13), c;
int b = 78;
c = a + b;   // CTiempo + int
Console.WriteLine(c.ToString());
```

Al compilar este código, se muestra un error que indica que el operador + no se puede aplicar a operandos de tipo **CTiempo** y **int**. Pues esto es porque no existe una sobrecarga del operador + que tenga un primer parámetro de tipo **CTiempo** y un segundo de tipo **int**. Añadamos la siguiente sobrecarga del operador +.

```
public static CTiempo operator +(CTiempo x, int y)
{
      CTiempo temp = new CTiempo(x.horas, x.minutos, x.segundos + y);
      if (temp.segundos >= 60)
      {
            temp.segundos -= 60;
            temp.minutos++;
      }
```

```
      if (temp.minutos >= 60)
      {
            temp.minutos -= 60;
            temp.horas++;
      }
            return temp;
}
```

Después de añadir el método anterior, todo funciona correctamente.

3.6.6 Sobrecarga de operadores unarios

Los operadores unarios o unitarios son aquellos que trabajan sobre un único operando. La sobrecarga es similar a la de un operador binario. Los operadores ++ y – – son los únicos operadores que se pueden usar como prefijo o como sufijo sobre un operando. Y cuando se utilizan el resultado se asigna a una variable, el valor es diferente dependiendo de cómo se haya utilizado.

Ejemplo:
El siguiente código muestra la sobrecarga del operador ++, el método correspondiente devolverá el objeto CTiempo para el que fue invocado incrementado en una unidad.

```
public static CTiempo operator ++(CTiempo q)
{
      CTiempo temp = new CTiempo(q.horas, q.minutos, q.segundos);
      temp.segundos += 1;
      if (temp.segundos >= 60)
      {
            temp.segundos -= 60;
            temp.minutos++;
      }
      if (temp.minutos >= 60)
      {
            temp.minutos -= 60;
            temp.horas++;
      }
      return temp;
}
```

En el código siguiente se observa que si el operador ++ se utiliza como sufijo, el valor de **a** será incrementado en una unidad, pero el método **operator ++** devolverá el valor de **c** sin incrementar. Y si el operador ++ se utiliza como prefijo, el valor de **a** será incrementado en una unidad y que **c** almacenara ese valor incrementado.

```
public class Test
{
      public static void Main(string[] args)
      {
            // . . .
            Console.WriteLine();
            // el operador ++ como sufijo
            CTiempo a = new CTiempo(4, 32, 17), c;
```

```
            c = a++;
            Console.WriteLine("la hora es : " + c.ToString());
            Console.WriteLine("la hora incrementada es : " +
a.ToString());
            // el operador ++ como prefijo
            c = ++a;
            Console.WriteLine("la hora es : " + c.ToString());
            Console.WriteLine("la hora incrementada es : " +
a.ToString());
      }
}
```

3.6.7 Operadores unarios/binarios

Un operador como – puede usarse indistintamente como operador unario o como operador binario. El método **operator –** como operador unario no tiene parámetros explícitos y devuelve un objeto CTiempo del mismo valor que el que invoca al método, pero de signo contrario.

Por ejemplo:

```
public static CTiempo operator -(CTiempo x)
{
      return new CTiempo(- x.horas, x.minutos, x.segundos);
}
```

```
public static void Main(string[] args)
{
    CTiempo hora = new CTiempo(), horaneg;
      hora.AsignarTiempo(7, 35, 40);
      // ...
      Console.WriteLine(hora.ToString());
      horaneg = -hora;
      Console.WriteLine(horaneg.ToString());
      //...
      Console.WriteLine();
}
```

3.6.8 Conversiones personalizadas

Hay dos tipos de conversiones:

- implícitas
- explícitas

Implícitas, las cuales son realizadas automáticamente por el compilador.
Explícitas, las cuales fuerzan una determinada conversión utilizando una construcción *cast*.

Cuando trabajamos con clases, por tratarse de tipos definidos por el usuario, tenemos que construir nosotros mismos las conversiones implícitas y/o explícitas que deseamos que el compilador realice cuando utilice un objeto de alguna de esas clases. Estas conversiones pueden ser entre clases o estructuras, o entre una clase o estructura y un tipo predefinido.

Para declarar un operador de conversión en una declaración de una clase o estructura se utiliza la palabra clave **operador**.

```
public static {implicit|explicit} operator tipo1(tipo2 objeto2)
{
    tipo1 objeto1 = new tipo1();
    // codigo para convertir de tipo2 a tipo1
    return objeto1;
}
```

3.6.9 Indexación

El operador de indexación, [], utilizado en matrices, no se puede sobrecargar; no obstante, los tipos pueden definir indizadores y propiedades que aceptan uno o varios parámetros. Los parámetros de un indicador van entre corchetes, como los índices de una matriz, pero se pueden declarar de cualquier tipo, a diferencia de los índices de una matriz, que sólo pueden ser de tipo integral.

Una definición de un indizador tiene la forma siguiente:

```
public int this[int ind] //Declaracion de un indizador
{
    get
    {
        //...
        return nombre_matriz[ind];
    }
    set
    {
        //...
        nombre_matriz[ind] = value;
    }
}
```

3.7 UNIDAD V: Herencia, Polimorfismo e interfaces

Titulo del tema:

Herencia, Polimorfismo e interfaces

Objetivos:

- Entender el concepto de herencia y polimorfismo.
- Entender el concepto de métodos virtuales e interfaces.

Contenido:

- Concepto de herencia
- Definir una clase derivada
- Control de acceso a los miembros de las clases
- Atributos con el mismo nombre
- Redefinir métodos de la clase base
- Constructores de las clases derivadas
- Copia de objetos
- Destructores de las clases derivadas
- Jerarquía de clases
- Métodos virtuales
- Polimorfismo
- Clases y métodos abstractos
- Interfaces
- Interfaces frente a herencia múltiple

Duración:

- 8 horas

Bibliografía básica:

- Ceballos Sierra, Francisco Javier. Microsoft C# Curso de Programación. Editorial RAMA.

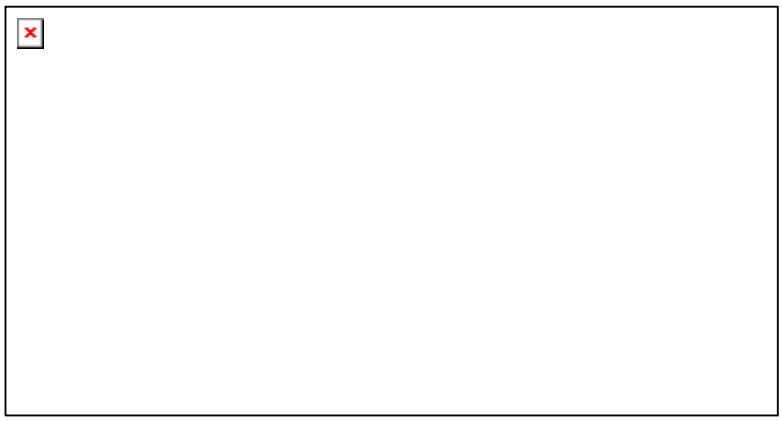
3.7 UNIDAD V: HERENCIA, POLIMORFISMO E INTERFACES

3.7.1 Concepto de herencia

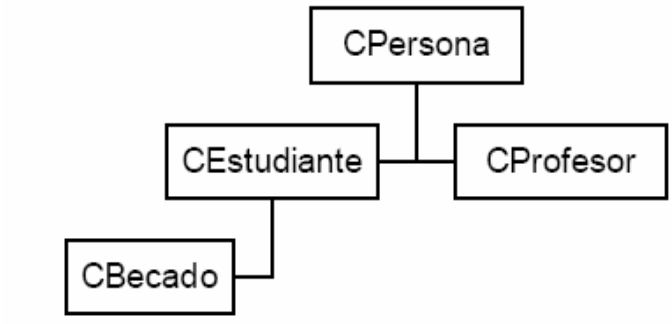
La herencia provee el mecanismo más simple para especificar una forma alternativa de acceso a una clase existente, o bien para definir una nueva clase que añada nuevas características a una clase existente. Esta nueva clase se denomina *clase derivada* o *subclase* y la clase existente, *clase base* o *superclase*.

Con la herencia todas las clases están clasificadas en una jerarquía estricta. Cada clase tiene su superclase (la clase superior en la jerarquía), y cada clase puede tener una o más subclases (las clases inferiores en la jerarquía).

En general, podemos tener una gran jerarquía de Clases tal y como vemos en el siguiente gráfico:



Las clases que están en la parte inferior en la jerarquía se dicen que *heredan* de las clases que están en la parte superior en la jerarquía. La herencia es una forma sencilla de reutilizar código. Para ilustrar el mecanismo de herencia vamos a implementar la jerarquía de clases siguiente:



En el ejemplo queremos declarar una serie de clases que describan personas, por ejemplo CEstudiante o CProfesor, y de la clase CEstudiante describir una clase CBecado. Pero estas clases tienen ciertos rasgos en común: el nombre, el documento único de identidad

DUI, la dirección, la edad. Vamos agrupar estas cosas en una clase CPersona de la cual posteriormente derivaremos las clases CEstudiante y CProfesor.

```
//Clase CPersona: clase que agrupa los datos comunes
//a cualquier persona.
public class CPersona
{
    //Atributos
    private string nombre;
    private string DUI;
    private string direccion;

    //metodos
    public CPersona() { }

    public CPersona(string nom, string dui, string dir)
    {
        AsignarNombre(nom);
        AsignarDUI(dui);
        AsignarDir(dir);
    }

    public bool AsignarNombre(string nom)
    {
        if (nom.Length == 0)
        {
            Console.WriteLine("Error : cadena vacia");
            return false;
        }
        nombre = nom;
        return nom.Length != 0;
    }

    public bool AsignarDUI(string dui)
    {
        if (dui.Length == 0)
        {
            Console.WriteLine("Error : cadena vacia");
            return false;
        }
        DUI = dui;
        return dui.Length != 0;
    }

    public bool AsignarDir(string dir)
    {
        if (dir.Length == 0)
        {
            Console.WriteLine("Error : cadena vacia");
            return false;
        }
        direccion = dir;
        return dir.Length != 0;
    }
    public string ObtenerNombre()
    {
```

```
        return nombre;
    }

    public string ObtenerDUI()
    {
        return DUI;
    }

    public string ObtenerDir()
    {
        return direccion;
    }
}
```

3.7.2 Definir una clase derivada

Continuando con nuestro ejemplo, vamos a diseñar una nueva clase CEstudiante, que tenga los mismos atributos y métodos de CPersona, a los que añadiremos los nuevos atributos y métodos propios de un estudiante como por ejemplo el número de carnet.
Así:

```
public class CEstudiante
{
    //Atributos y metodos de CPersona
    //Nuevos Atributos y metodos de CEstudiante
}
```

```
public class CEstudiante
{
    CPersona per = new CPersona();
    // Nuevos atributos y metodos de CEstudiante
}
```

Al hacer esto supone un derroche de tiempo y esfuerzo, todo lo realizado antes no ha servido de nada.
No hay nada que indique al compilador que un objeto **CEstudiante** es un objeto **CPersona**, y tampoco un referencia a **CEstudiante** es una referencia a **CPersona**; hay que especificar de manera explícita que **CEstudiante** es una extensión **CPersona**. Esto se hace a través de la herencia, definiendo una clase derivada de la clase existente.

Una *clase derivada* es un nuevo tipo de objetos definido por el usuario que tiene la propiedad de heredar los atributos, propiedades y métodos de otra clase definida previamente, denominada *clase base*.

La sintaxis para definir una clase derivada es la siguiente:

```
class ClaseDerivada : ClaseBase
{
      //Cuerpo de la clase derivada
}
```

Ejemplo:

```
public class CEstudiante : CPersona
{
    //CEstudiante ha heredado los miembros de CPersona
    //Nuevos atributos y metodos de CEstudiante
}
```

Una clase derivada puede, a su vez, ser una clase base de otra clase, dando lugar a una *jerarquía de clases*.
Cuando una clase derivada lo es de una sola clase base, la herencia se denomina *herencia simple* o *derivación simple*. En cambio, cuando lo es de dos o más clases, la herencia se denomina *múltiple* o *derivación múltiple*. C# no permite la herencia múltiple.

3.7.3 Control de acceso a los miembros de las clases

Anteriormente se mencionó que para controlar el acceso a los miembros de una clase, C# provee las palabras clase **private**, **protected**, **public**, **internal** y **protected internal**.
La tabla siguiente resume de una forma clara qué clases, o clase derivadas, pueden acceder a los miembros de otra clase, dependiendo del control de acceso especificado.

Puede ser accedido desde:	Un miembro declarado en una clase como			
	privado	interno	protegido	público
Su misma clase …….	Si	Si	Si	Si
Cualquier clase del mismo ensamblado	No	Si	No	Si
Cualquier clase derivada del mismo ensamblado ……..	No	Si	Si	Si
Cualquier clase de otro ensamblado ……	No	No	No	Si
Cualquier clase derivada de otro ensamblado ……..	No	No	Si	Si

Miembros que hereda una clase derivada

- Una clase derivada hereda todos los miembros de su clase base, excepto los constructores y destructores.
- Una clase derivada no tiene acceso directo a los miembros privados (**private**) de su clase base, pero sí puede acceder a los miembros públicos (**public**), protegidos (**protected**) y a los miembros internos (**internal**) de su clase base.
- Una clase derivada puede añadir sus propios atributos y métodos. Si el nombre de alguno de estos miembros coincide con el de un miembro heredado, este último queda oculto para la clase derivada, que se traduce en que la clase derivada ya no puede acceder directamente a ese miembro.
- Los miembros heredados por una clase derivada pueden, a su vez, ser heredados por más clases derivadas de ella. A esto se le llama propagación de la herencia.

Continuando con nuestro ejemplo diseñemos una nueva clase **CEstudiante** que tenga, además las capacidades de **CPersona**, otras como las siguientes:

```
public class CEstudiante : CPersona
{
    //Atributos
    private string N_Carnet;

    //Metodos
    public CEstudiante()
    {
    } //constructor sin parametros

    public bool AsignarN_Carnet(string carnet)
      {
            if (carnet.Length == 0)
            {
                  Console.WriteLine("Error: cadena vacia");
                  return false;
            }
            N_Carnet = carnet;
            retrun carnet.Length != 0;
      }

    private string ObtenerNumCarnet()
    {
        return N_Carnet;
    }
}
```

CEstudiante es una clase derivada de la clase base **CPersona**.
La clase CPersona define sus constructores, pero aunque no se definieran, el compilador generaría uno por omisión. Esto implica que los constructores no se heredan. Y tampoco se heredaría el destructor si se escribiera.

Escribamos una pequeña aplicación basada en una clase *Test* que cree un objeto **CEstudiante**.

```
public class Test
{

    public static void Main(string[] args)
    {
        CEstudiante per = new CEstudiante();
        per.AsignarNombre("Gustavo");
        per.AsignarDUI("02339024-5");
        per.AsignarDir("San Miguel, El Salvador");
        per.AsignarN_Carnet("MD-23098");
        Console.WriteLine(per.ObtenerNombre());
        Console.WriteLine(per.ObtenerDUI());
        Console.WriteLine(per.ObtenerDir());
        Console.WriteLine(per.ObtenerNumCarnet());
    }
}
```

3.7.4 Atributos con el mismo nombre

Una clase derivada puede acceder directamente a un atributo público, protegido, o interno en su caso, de su clase base. ¿Qué sucede si se define en la clase derivada uno de estos atributos con el mismo nombre que tiene en la clase base?

Ejemplo:

```
public class ClaseX
{
    public int atributo_a = 1;

    public int metodo_a()
    {
        return atributo_a * 5;
    }

    public int metodo_b()
    {
        return atributo_a + 7;
    }
}

public class ClaseY : ClaseX
{
    public int atributo_a = 2;

    public int metodo_a()
    {
        return atributo_a * -5;
    }
}
```

La definición del atributo *atributo_a* en la clase derivada oculta la definición del atributo con el mismo nombre en la clase base. Es así como las referencias a *atributo_a* en el código del ejemplo siguiente devolverán el valor de *atributo_a* de ClaseY. Si este atributo no hubiera sido definido en la clase derivada, entonces el valor devuelto sería el valor de *atributo_a* de la clase base.

```
public class Test
{
    public static void Main(string[] args)
    {
        ClaseY objClaseY = new ClaseY();
        Console.WriteLine(objClaseY.atributo_a);    //escribe 2
        Console.WriteLine(objClaseY.metodo_b());    //escribe 8
        Console.WriteLine(objClaseY.metodo_a());    //escribe -10
    }
}
```

Si el *método_a* de la clase ClaseY tuviera que acceder obligatoriamente al dato *atributo_a* de la clase base, se procede a utilizar para ese atributo nombres diferentes en la clase base y en la clase derivada. Aunque al utilizar el mismo nombre, se tiene la alternativa de acceso: utilizar la palabra reservada **base**. Por ejemplo:

```
public int metodo_a()
{
    return base.atributo_a * -5;
}
```

Se puede referir al dato *atributo_a* de la clase base y la clase derivada, con expresiones *base.atributo_a* y *this.atributo_a* respectivamente. La expresión *((ClaseX)this).atributo_a* hace referencia al dato *atributo_a* de la clase X.

3.7.5 Redefinir métodos de la clase base

Redefinir un método heredado significa volverlo a escribir en la clase derivada con el mismo nombre, la misma lista de parámetros y el mismo tipo del valor retornado que tenía en la clase base; su cuerpo será adaptado a las necesidades de la clase derivada. Esto es lo que se ha hecho con el *método_a* del ejemplo anterior.

El *método_a* ha sido redefinido en la ClaseY para que realice unos cálculos diferentes a los que realizaba en la ClaseX.

Cuando en una clase derivada se redefine un método de una clase base, se oculta el método de la clase base, pero no las sobrecargas que existan del mismo en dicha clase base. Por otra parte si el método se redefine en la clase derivada con distinto tipo o número de parámetros, el método de la clase base no se oculta, sino que se comporta como una sobrecarga de ese método.

Así,

```
public int metodo_a(int c)    //metodo de la ClaseY
{
    return atributo_a * -c;
}
```

Para acceder a un método de la clase base que ha sido redefinido en la clase derivada, igual que se dijo para los atributos, se debe de utilizar la palabra **base**.

Ejemplo:

```
public int metodo_muestra()
{
    atributo_a = base.atributo_a + 10;
    return base.metodo_a() + atributo_a;
}
```

En el fragmento anterior se hace referencia al *método_a* de la clase base y si se quiere hacer referencia al *método_a* de la clase derivada se utiliza **this**, de la siguiente forma:

```
this.metodo_a()
```

3.7.6 Constructores de las clases derivadas

Cuando se crea un objeto de una clase derivada, se invoca a su constructor, que a su vez invoca al constructor sin parámetros de la clase base, que a su vez invoca al constructor de su clase base, y así sucesivamente. Esto es porque una clase derivada contiene todos los atributos de su clase base, y todos tienen que ser iniciados, razón por la que el constructor de la clase derivada tiene que llamar implícita o explícitamente al de la clase base.

Para permitir que el constructor de la clase derivada invoque explícitamente al constructor de la clase base para crear objetos de la clase derivada, se usa la siguiente sintaxis:

```
public nombre_clase derivada(parametros-d) : base(parametros-b)
{
    //cuerpo del constructor de la clase derivada
}
```

No especificar una llamada explícita al constructor de la clase base equivale a:

```
public nombre_clase derivada(parametros-d) : base()
{
    //cuerpo del constructor de la clase derivada
}
```

El orden de ejecución es:

1. Se construyen los atributos del objeto de la clase derivada (los propios y los heredados).
2. Constructor de la clase base.
3. Cuerpo del constructor de la clase derivada.

El orden en el que se ejecutan los pasos anteriores se aplican recursivamente para cada constructor de cada una de las clases.

Continuando con el ejemplo de la clase **CPersona**, vamos a añadir a la clase **CEstudiante** un constructor con parámetros. Este constructor debe de tener tantos parámetros como atributos heredados y propios tenga la clase. Para el ejemplo que continuamos trabajando la clase **CEstudiante** contiene los atributos *nombre, DUI, dirección* y *N_Carnet*.

Así,

```
public CEstudiante(string nom, string dui, string dir, string car):
base(nom, du, dir)
{
      AsignarN_Carnet(car);
}
```

Con la palabra **base** del constructor llama al constructor de **CPersona**, clase base de **CEstudiante**. La clase **CPersona** debe tener un constructor con tres parámetros del tipo de los argumentos especificados. Después, el cuerpo del constructor invoca al método *AsignarN_Carnet* para iniciar el atributo N_Carnet de **CEstudiante**.

De acuerdo con los constructores definidos en la clase **CEstudiante,** son declaraciones válidas las siguientes:

```
public class Test
{

    public static void Main(string[] args)
    {
        CEstudiante per = new CEstudiante();
        CEstudiante per1 = new CEstudiante("Hugo", "023-34254-3", "San
Andrew", "WE-34254");
        //...
    }
}
```

La primera sentencia requiere en CEstudiante un constructor sin parámetros y en CPersona otro, y la segunda sentencia requiere en CEstudiante un constructor con parámetros y en CPersona otro que se pueda invocar con el fin de iniciar los atributos definidos en la clase con los valores pasados como argumentos, así como se indica a continuación:

base (nombre, dui, dirección);

Cuando se crea *per* o *per1*, primero se construye la porción del objeto correspondiente a su clase base y a continuación la porción del objeto correspondiente a su clase derivada.

3.7.7 Copia de objetos

Esto es escribir un constructor copia y/o un método *copiar*. Como una clase derivada contiene todos los atributos de su clase base, y todos tienen que ser copiados, la mejor solución es definir estos métodos tanto en las clase base como en las clases derivadas para que éstas puedan invocar a las versiones de sus clases base.

El constructor copia y el método *Copiar* de la clase CPersona es como se muestra a continuación:

```
public CPersona(CPersona A)
{
      Copiar(A);
}

public CPersona Copiar(CPersona A)
{
      nombre = A.nombre;
      DUI = A.DUI;
```

```
      direccion = A.direccion;
      return this;
}
```

El constructor copia y el método *Copiar* de la CEStudiante es como se muestra a continuación:

```
public CEstudiante(CEstudiante N_C):base(N_C)
{
      N_Carnet = N_C.N_Carnet;
}

public CEstudiante Copiar(CEstudiante N_C)
{
      base.Copiar(N_C);
      N_Carnet = N_C.N_Carnet;
      return this;
}
```

Son declaraciones válidas las siguientes:

```
public static void Main(string[] args)
{
      CEstudiante per = new CEstudiante();
      CEstudiante per1 = new CEstudiante("Hugo", "023-34254-3", "San
Andrew", "WE-34254");
      //...
      per.Copiar(per1);
      CEstudiante per2 = new CEstudiante(per1);
    //...
}
```

```
CEstudiante per2 = new CEstudiante(per1);
```

La línea anterior cuando se ejecuta ocurre lo siguiente:

1. Se invoca al constructor copia CEstudiante pasando como argumento el objeto *per1*. Su parámetro N_C referencia a este objeto.
2. Se invoca al constructor copia CPersona pasando como argumento el objeto CEstudiante referenciado por N_C. Su parámetro A referencia a este objeto. Pero A es una referencia a un objeto CPersona y todo ha funcionado correctamente. Debido a que existe una conversión implícita de referencias a objetos de la clase derivada a sus correspondientes de la clase base.

3.7.8 Destructores de las clases derivadas

El destructor de una clase base no es heredado por sus clases derivadas. Los objetos de las clases derivadas se destruyen en el orden inverso a como son construidos.
El destructor de la clase **CPersona** y de la clase **CEstudiante** está definido de la forma siguiente:

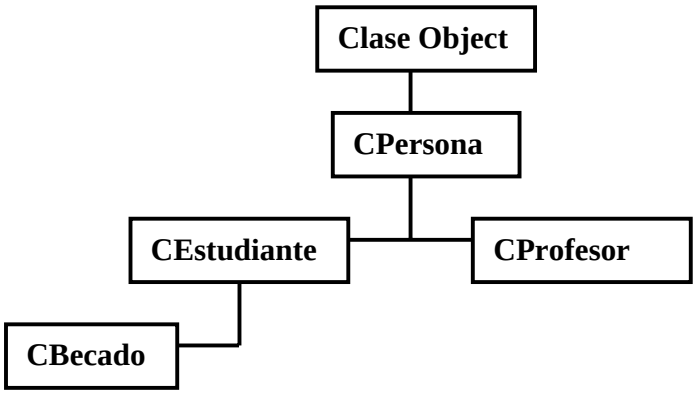
```
~CPersona(){}
```

```
~CEstudiante(){}
```

Siempre que se destruya un objeto de la clase CEstudiante, primero se ejecutarán los destructores de sus atributos, después se ejecutará el cuerpo de ~CEstudiante y por último se ejecutará el destructor de ~CPersona.

3.7.9 Jerarquía de clases

Una clase derivada puede asimismo ser una clase base de otra clase, y así sucesivamente. Como puede verse en la siguiente figura.



Este esquema da lugar a una *jerarquía de clases*. Cuando cada clase derivada lo es de una sola clase base, la estructura jerárquica recibe el nombre de **árbol de clases**.

De la misma manera como se diseño la clase derivada CEstudiante se procede a diseñar las demás clases derivadas. Para implementar una clase derivada como por ejemplo **CBecado** basta con conocer su clase base **CEstudiante** y no interesa **CPersona**.

3.7.10 Métodos virtuales

Cuando se invoca a un método que está definido en la clase base y redefinido en sus clases derivadas, la versión que se ejecuta depende del tipo de la referencia que se utilice para invocar al mismo.

Un método *virtual* es un miembro de una clase base que puede ser redefinido (**override**) en cada una de las clase derivadas de ésta, y una vez redefinido puede ser accedido mediante una referencia a la clase base, resolviéndose la llamada en función del tipo del objeto referenciado. Una clase con métodos virtuales se denomina *tipo polimórfico*.

Un método se declara *virtual* escribiendo el modificador **virtual** en la declaración del método en la clase base:

public virtual *tipo nombre_método(parámetros)*
{
    // Cuerpo del método

}

Cuando se declara un método virtual, toda clase que hereda el método puede implementar su propia versión.

Los métodos virtuales deben ser redefinidos en las clases derivadas utilizando el modificador **override** u ocultar el método virtual de la clase base con la palabra clase **new**.

public override *tipo nombre_método(parámetros)*
{
    //Cuerpo del método
}

La redefinición de un método virtual en una clase derivada debe tener el mismo nombre, número y tipos de parámetros, y el tipo del valor retornado que en la clase base, de lo contrario se producirá un error.

Para mostrar lo antes mencionado, vamos a utilizar una clase base llamada **CCelular**, cuyo código se muestra a continuación:

```
public class CCelular
{
    //Atributos
    private string marca;

    //Metodos
    public CCelular()
    {
    }

    public CCelular(string mar)
    {
        AsignarMarca(mar);
    }
    public virtual void AsignarMarca(string mar) //metodo virtual
    {
        if (mar.Length == 0)
        {
            System.Console.WriteLine("Error: cadena vacia ");
            return;
        }
        marca = mar;
    }

    public string obtenerMarca()
    {
        return marca;
    }
}
```

A continuación se muestra la clase **CCelularNuevo** derivada de **CCelular:**

```
class CCelularNuevo : CCelular
{
    //Atributos
    private double precio_Venta;

    //Metodos
    public CCelularNuevo()
    { //constructor sin parametros
    }

    public CCelularNuevo(string mar, double preVent)
        : base(mar)
    {
        asignarPrecioVenta(preVent);
    }

    public void asignarPrecioVenta(double cantidad)
    {
        if (cantidad < 0)
            System.Console.WriteLine("Error: Cantidad negativa");
        else
            precio_Venta = cantidad;
    }

    public double obtenerPrecioVenta()
    {
        return precio_Venta;
    }

    public override void AsignarMarca(string mar) //metodo virtual
redefinido
    {
        System.Console.WriteLine("Esta es la marca de mi Nuevo Celular
" + mar);
        base.AsignarMarca(mar); //metodo AsignarMarca de la clase base
    }
}
```

Ahora se muestra la clase **Test:**

```
public class Test
{
    public static void Main(string[] args)
    {
        CCelular miCelular = new CCelular();

        miCelular.AsignarMarca("Motorola");
        System.Console.WriteLine("La marca del celular es : " +
miCelular.obtenerMarca());

        CCelularNuevo miOtroCelular = new CCelularNuevo();

        miOtroCelular.AsignarMarca("Sony Ericsson"); //metodo
redefinido (el heredado ha quedado oculto)

        miOtroCelular.asignarPrecioVenta(25.35);
```

```
        System.Console.WriteLine("El precio del celular nuevo es : $ "
+ miOtroCelular.obtenerPrecioVenta());
    }
}
```

Los resultados que produce el programa anterior son los siguientes:

```
La marca del celular es : Motorola
Esta es la marca de mi Nuevo Celular Sony Ericsson
El precio del celular nuevo es : $ 25.35
```

Se observa que es el método `AsignarMarca` *()* de la clase `CCelularNuevo` el que se ejecuta al recibir el mensaje del objeto `miOtroCelular`, ya que el método de la clase base ha quedado oculto.

```
CCelularNuevo miOtroCelular = new CCelularNuevo();
miOtroCelular.AsignarMarca("Sony  Ericsson");  //metodo  redefinido(el
heredado ha quedado oculto)
```

Conclusiones sobre los resultados anteriores:

- Una llamada a un método virtual se resuelve siempre en función del tipo del objeto referenciado.

- Una llamada a un método normal (no virtual) se resuelve en función del tipo de la referencia.

Cuando la clase derivada no redefine el método virtual de su clase base, hereda la implementación de la clase base, de manera que una llamada al mismo a través de un objeto de la clase derivada hace que se ejecute el método virtual de su clase base.

3.7.11 Polimorfismo

Se denomina *Polimorfismo c*onseguir que los métodos de una clase base y sus redefiniciones en sus clases derivadas se comporten adecuadamente, independientemente del tipo del medio realmente empleado para acceder a los mismos (referencia a una clase). Se refiere a la facultad de asumir muchas formas.

Los métodos deben ser definidos virtuales (**virtual**) en la clase base y redefinidos (**override**) en las clases derivadas, y los objetos deben ser manipulados mediante referencias a la clase base. Si se dan estas condiciones, cuando se invoque a uno de esos métodos, la versión que se ejecutará corresponderá a la clase del objeto referenciado y no a la clase de la variable que lo referencia.

¿Cómo implementamos en C# el polimorfismo?. Muy fácil. Ya lo hemos hecho en el ejemplo anterior mediante los métodos virtuales y su posterior redefinición con los de

reemplazo. Haciendo algunos cambios a la clase que se ha venido trabajando, se vera el siguiente ejemplo:

```
public class CCelular
{
    //Atributos
    public string marca;

    //metodo virtual
    public virtual void AsignarMarca()
    {
        System.Console.WriteLine(marca);
    }
}

public class CCelularNuevo : CCelular
{
    //atributos
    public double precio_Venta;

    //metodo de reemplazo
    public override void AsignarMarca()
    {
        System.Console.WriteLine("La marca de mi Nuevo Celular es " +
marca + " y el precio de Venta es :$ " + precio_Venta);
    }
}

public class CCelularArruinado : CCelular
{
    //atributos
    public string motivo;

    //metodo de reemplazo
    public override void AsignarMarca()
    {
        System.Console.WriteLine("La marca del Celular Arruinado es " +
marca + ", y el motivo es porque " + motivo);
    }
}

public class Test
{
    public static void Main(string[] args)
    {
        CCelularNuevo CelularNuevo = new CCelularNuevo();
        CelularNuevo.marca = "Alcatel";
        CelularNuevo.precio_Venta = 80;

        CCelularArruinado celularArr = new CCelularArruinado();
        celularArr.marca = "Sanyo";
        celularArr.motivo = "No carga bateria";

        CCelular miCelular;         //miCelular es una referencia, no
un objeto!
```

```
        miCelular = CelularNuevo;   //miCelular referencia a un Celular
nuevo (clase derivada)
        miCelular.AsignarMarca();   //actua el metodo de reemplazo de
CCelularNuevo

        miCelular = celularArr;           //ahora miCelular referencia
a un Celular Arruinado
        miCelular.AsignarMarca();   //actua el metodo de reemplazo de
CCelularArruinado
        System.Console.ReadLine();
    }
}
```

La salida en pantalla de este código es:

```
La marca de mi Nuevo Celular es Alcatel y el precio de Venta
es :$ 80
La marca del Celular Arruinado es Sanyo, y el motivo es
porque No carga bateria
```

Justo lo que se quería. Si `miCelular`, que es una referencia de tipo `CCelular`, referencia un objeto de la clase derivada `CCelularNuevo` el método que se activa al ejecutar `miCelular.AsignarMarca()` es el redefinido para `CCelularNuevo`. Si referencia a un objeto de la clase `CCelularArruinado`, el método que se activa al invocar `miCelular.AsignarMarca()` es el redefinido para la clase `CCelularArruinado`.

3.7.12 Clases y métodos abstractos

Cuando una clase se diseña para ser genérica, es casi seguro que no necesitamos crear objetos de ella; la razón de su existencia es proporcionar los atributos y comportamientos que serán compartidos por todas sus clases derivadas.

Una clase que se comporta de la forma descrita se denomina clase *abstracta* y se define como tal calificándola explícitamente *abstracta* (**abstract**).

Por ejemplo:

```
public abstract class MiClaseAbs
{
    //Cuerpo de la clase
}
```

Los métodos abstractos son también por definición métodos virtuales y debe por lo tanto usarse la palabra clave ***override*** para reemplazarlos en las clases derivadas.

Modificar el ejemplo anterior de forma que la clase `CCelular` se va a declarar abstracta. El método `AsignarMarca()` se va a marcar como abstracto. Esto implicará que no podamos declarar objetos de la clase `CCelular` (realmente lo que nos interesa es

trabajar con Celulares nuevos y Celulares arruinados); aunque si podremos declarar referencias de `CCelular`. Por otro lado deberemos obligatoriamente implementar el método `AsignarMarca()` en las clases derivadas ya que en la clase `CCelular` no se implementa. Quedaría así:

```
public abstract class CCelular     //clase abstracta
{
    //Atributos
    public string marca;

    //metodo abstracto
    public abstract void AsignarMarca();

}

class CCelularNuevo : CCelular
{
    //atributos
    public double precio_Venta;

    //metodo de reemplazo
    public override void AsignarMarca()
    {
        System.Console.WriteLine("La marca de mi Nuevo Celular es " +
marca + " y el precio de Venta es :$ " + precio_Venta);
    }
}

class CCelularArruinado : CCelular
{
    //atributos
    public string motivo;

    //metodo de reemplazo
    public override void AsignarMarca()
    {
        System.Console.WriteLine("La marca del Celular Arruinado es " +
marca + ", y el motivo es porque " + motivo);
    }
}

class Test
{
    public static void Main(string[] args)
    {
        CCelularNuevo CelularNuevo = new CCelularNuevo();
        CelularNuevo.marca = "Alcatel";
        CelularNuevo.precio_Venta = 80;

        CCelularArruinado celularArr = new CCelularArruinado();
        celularArr.marca = "Sanyo";
        celularArr.motivo = "No carga bateria";

        CCelular miCelular;         //miCelular es una referencia, no
un objeto!
```

```
        miCelular = CelularNuevo;   //miCelular referencia a un Celular
nuevo (clase derivada)
        miCelular.AsignarMarca();   //actua el metodo de reemplazo de
CCelularNuevo

        miCelular = celularArr;           //ahora miCelular referencia
a un Celular Arruinado (clase derivada)
        miCelular.AsignarMarca();   //actua el metodo de reemplazo de
CCelularArruinado
        System.Console.ReadLine();
    }
}
```

La salida en pantalla de este código es:

```
La marca de mi Nuevo Celular es Alcatel y el precio de Venta
es :$ 80
La marca del Celular Arruinado es Sanyo, y el motivo es
porque No carga bateria
```

3.7.13 Interfaces

Una interfaz se define así: un dispositivo o un sistema utilizado por entidades inconexas para interactuar. Un control remoto es una interfaz, el idioma inglés es una interfaz, etc. Una interfaz C# es un dispositivo que permite interactuar a objetos no relacionados entre sí.

Las interfaces C# en realidad definen un conjunto de mensajes que se puede aplicar a muchas clases de objetos, a los que cada una de ellas debe responder de forma adecuada.

Una interfaz consta de dos partes:

- El *nombre de la interfaz* precedido por la palabra clave **interface**,
- El *cuerpo de la interfaz* encerrado entre llaves.

```
[modificadores] interface nombre_interfaz[:interfaces-base]
{
    //Cuerpo de la interfaz
}
```

El cuerpo de la interfaz sólo puede incluir declaraciones de métodos (no sus definiciones). El símbolo : significa que se está definiendo una interfaz que es una extensión de otras. A diferencia de las clases, una interfaz puede derivarse de más de una interfaz base.

El nombre de una interfaz se puede utilizar en cualquier lugar donde se puede utilizar el nombre de una clase.

Las interfaces, al igual que las clases y métodos abstractos, proporcionan plantillas de comportamiento que se espera sean implementadas por otras clases. Esto es, una interfaz

C# declara un conjunto de métodos, pero no los define (sólo aporta los prototipos de los métodos). Una interfaz no puede contener atributos (campos).

Ejemplo:

Vamos a realizar un ejemplo para una interfaz *IFecha*.

```
//Interfaz IFecha: metodos para obtener el dia, mes y año
public interface IFecha
{
    int dia();
    int mes;
    int año();
}
```

Para utilizar una interfaz hay que añadir el nombre de la misma de la siguiente forma:

```
public class Clase: Interfaz
public class ClaseDerivada: ClaseBase, Interfaz
public class Clase: Interfaz1, Interfaz2
...
```

Siguiendo con el ejemplo iniciado en el apartado anterior, la clase **CCelular** utiliza la interfaz *IFecha*, para mostrar la fecha de adquisición de un celular, así:

```
using System;

namespace PruebaInterfaces
{

    public interface IFecha
    {
        int dia();
        int mes();
        int anyo();
    }

    public class CCelular : IFecha
    {
        //Atributos
        private string marca;

        //Metodos
        public CCelular()
        {
        }

        public CCelular(string mar)
        {
            AsignarMarca(mar);
        }

        public int dia()
        {
```

```
            return DateTime.Now.Day;
        }
        public int mes()
        {
            return DateTime.Now.Month;
        }

        public int anyo()
        {
            return DateTime.Now.Year;
        }

        //...
    }

    public class CCelularNuevo : CCelular
    {
        //Atributos
        private double precio_Venta;

        //...
    }
    public class Test
    {
        public static void Main(string[] args)
        {
            CCelular miCelular = new CCelular();

System.Console.WriteLine("La fecha de adquisicion es: " +
miCelular.dia() + "-" + miCelular.mes() + "-" + miCelular.anyo());
      //...
        }
    }
}
```

Como una interfaz sólo aporta declaraciones de métodos, es nuestra obligación definir todos los métodos en cada una de las clases que utilice la interfaz. No podemos elegir y definir sólo aquellos métodos que necesitemos. De no hacerlo, el compilador C# mostraría un error.

Si una clase implementa una interfaz, todas sus clases derivadas heredarán los nuevos métodos que se hayan implementado en la clase base, así como las constantes definidas por la interfaz.

3.7.14 Interfaces frente a herencia múltiple

Los conceptos de interfaz y herencia múltiple son bastantes diferentes. A menudo se piensa en las interfaces como una alternativa a la herencia múltiple.

- Desde una interfaz, una clase no puede heredar campos.
- Desde una interfaz, una clase no puede heredar definiciones de métodos.

- La jerarquía de interfaces es independiente de la jerarquía de clases. De hecho varias clases pueden implantar la misma interfaz y no pertenecer a la misma jerarquía de clases.
- En cambio, cuando se habla de herencia múltiple, todas las clases pertenecen a la misma jerarquía.

3.8 UNIDAD VI: EXCEPCIONES

Titulo del tema:

Excepciones

Objetivo:

- Conocer el manejo de situaciones anómalas que pueden ocurrir durante la ejecución de un programa, esto mediante el lanzamiento de excepciones.

Contenido:

- ¿Qué son las excepciones?
- Excepciones de C#
- Manejar excepciones
- Lanzar una excepción
- Capturar una excepción
- Excepciones derivadas
- Capturar cualquier excepción
- Relanzar una excepción
- Bloque de finalización
- Crear excepciones

Duración:

- 6 horas

Bibliografía básica:

- Ceballos Sierra, Francisco Javier. Microsoft C# Curso de Programación. Editorial RAMA.

3.8 UNIDAD VI: EXCEPCIONES

3.8.1 ¿Qué son las excepciones?

Durante la ejecución de un programa en C# pueden ocurrir situaciones anómalas, éstas situaciones son conocidas como "**excepciones**".

Las excepciones son manejadas por código fuera del flujo normal de control del programa. Y proporcionan una manera limpia de verificar errores.

Lo que se persigue es que el programa no sea abortado inesperadamente por el sistema, sino diseñar una continuación o terminación normal dentro de lo ocurrido, y se utilizan las sentencias **if** y **switch** para controlar los posibles errores que se puedan dar.

El manejo de excepciones ofrece una forma de separar explícitamente el código que maneja los errores del código básico de una aplicación, haciéndola más legible, lo que desemboca un estilo de programación.

```
try
{
    //Codigo de la aplicacion
}
catch(clase_de_excepcion e)
{
    //Codigo de tratamiento de esta excepcion
}
```

3.8.2 Excepciones de C#

Algunas excepciones son:

Clase de excepción	*Significado*
DivideByZeroException	Se produce cuando se intenta dividir un valor entero o decimal entre cero.
IndexOutOfRangeException	Una matriz fue accedida con un índice ilegal (fuera de los límites permitidos).
NullReferenceException	Se intentó utilizar **null** donde se requería un objeto.
FormatException	Se produce cuando el formato de un argumento no es el adecuado.

Las excepciones en C# son objetos de clases derivadas de la clase **Exception** definida en el espacio de nombres **System**.

La clase **Exception** cubre las excepciones que una aplicación normal puede manipular. Tiene varias clases derivadas entre las que destacan: **SystemException** y **ApplicationException**.

3.8.3 Manejar excepciones

El funcionamiento es el siguiente: cuando un método encuentra una anomalía que no puede resolver, *lanza* (**throw**) una excepción, esperando que quien la llamó directa o indirectamente *capture* (**catch**) la excepción y maneje la anomalía. Incluso el mismo método podría capturar y manipular la excepción. Si la excepción no se captura, el programa finalizará automáticamente.

Ejemplo:

```
using System;

namespace CFecha
{
    public class Leer
    {
        //...
        public static float datoFloat()
        {
            try
            {
                return Single.Parse(Console.ReadLine());
            }
            catch (FormatException)
            {
                return Single.NaN;  //No es un numero; valor float
            }
        }
        //...
    }
}
```

En el ejemplo anterior el método *datoFloat* de la clase Leer invoca al método **Parse** con el propósito de devolver el valor **float** correspondiente a la cadena leída. **Parse** lanza una excepción de la clase **FormatException** si la cadena de caracteres a analizar no se corresponde con un valor **float.** Para manejar esta situación hay que capturar la excepción cuando se lance, para lo cual se utiliza un bloque c**atch**, y para poder capturarla hay que encerrar el código que puede lanzarla en un bloque **try**.

3.8.4 Lanzar una excepción

Equivale a crear un objeto de la clase de la excepción para manipularlo fuera del flujo normal de ejecución del programa. Para lanzar una excepción se utiliza la palabra reservada **throw** y para crear un objeto, **new**.

Si ocurre un error en el método *datoFloat* en el ejemplo anterior, cuando se ejecute el método **Parse** se supone que éste ejecutará una sentencia similar a la siguiente:

```
if (error) throw new FormatException();
```

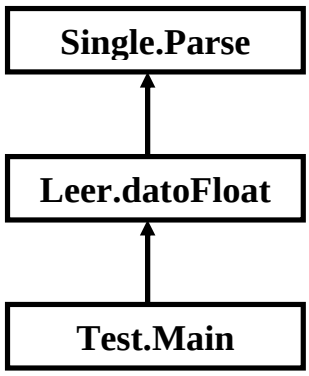
La sentencia anterior lanza una excepción de la clase **FormatException** lo que implica crear un objeto de esta clase.

3.8.5 Capturar una excepción

Una vez lanzada la excepción, el sistema es responsable de encontrar a alguien que la capture con el objetivo de manipularla.

```
public class Test
{
    public static void Main(string[] args)
    {
        float f;
        Console.Write("Dato: ");
        f = Leer.datoFloat();
        //...
    }
}
```

La aplicación anterior cuando se ejecute y se invoque al método *datoFloat*, la pila de llamadas crecerá como se observa en la figura siguiente:



Para implementar un manejador para una clase de excepción hay que hacer las dos cosas que se indican a continuación:

1. Encerrar el código que puede lanzar la excepción en un bloque **try**.

```
try
{
   return Single.Parse(Console.ReadLine()
}
```

2. Escribir un bloque **catch** capaz de capturar la excepción lanzada.

```
catch (FormatException e)
{
    Console.WriteLine("Error: " + e.Message);
    return Single.NaN;  //No es un numero; valor float
}
```

En este manejador se observa un parámetro *e* que referencia al objeto que se creó cuando se lanzó la excepción capturada. Este parámetro se puede omitir, el código sería el siguiente:

```
catch (FormatException)
{
    return Single.NaN;  //No es un numero; valor float
}
```

3.8.6 Excepciones derivadas

- Cuando se trata de manejar excepciones, un bloque **try** puede estar seguido de uno o más bloques **catch,** tantos como excepciones diferentes tengamos que manejar.

- Cada **catch** tendrá un parámetro de la clase **Exception**, de alguna clase derivada de ésta, o bien de una clase de excepción definida por el usuario.

- Cuando se lance una excepción, el bloque **catch** que la capture será aquél cuyo parámetro sea de la clase de la excepción o de una clase base directa o indirecta.

3.8.7 Capturar cualquier excepción

Especificar **catch** sin parámetros significa "capturar cualquier excepción". Un bloque **catch** que capture todas las excepciones será el siguiente:

```
//...
catch
{
    //finalizacion adecuada para el proceso en curso
    throw; //lanzar otra vez la excepcion
}
```

3.8.8 Relanzar una excepción

Después de haber capturado una excepción, el manejador puede volverla a lanzar; esto se hace ejecutando **throw** sin ningún operando. Esto puede ser útil en los casos en los que el manejador que capturó una excepción decida que no puede tratar completamente el error.

3.8.9 Bloque de finalización

Si no se trata de manejar excepciones, sino de realizar alguna acción necesaria después de salir del bloque **try**, hay que poner el código adecuado dentro de un bloque **finally** colocado después del bloque **try** o de un bloque **catch**. El bloque **finally** debe ser siempre el último.

La ejecución del bloque **finally** queda garantizada independientemente de que finalice o no la ejecución del bloque **try**.

```
try
{
    //codigo de try
}
catch(exception)
{
    //codigo de catch
}
finally
{
    //codigo de finally
}
```

3.8.10 Crear excepciones

En la biblioteca de clases de C# hay una gran cantidad de excepciones que podemos utilizar, pero se puede necesitar en algún momento crear nuestras propias excepciones. Cualquiera que sea el caso, todos los tipos de excepción se corresponden con una clase derivada de **Exception**, clase raíz de la jerarquía de clases de excepciones de C#.

Algunas de las propiedades de la clase **Exception** se presentan a continuación:

Data Permite obtener una colección de pares clave-valor que proporcionan información adicional definida por el usuario acerca de la excepción.

Message Permite obtener un mensaje de texto para informar de la naturaleza del error y recomendar alguna acción para solucionar el problema.

Source Devuelve o establece el nombre de la aplicación o del objeto que generó el error.

TargetSite Permite obtener el método que produce la excepción actual.

Cuando se quiera manejar un determinado tipo de error no contemplado por las excepciones proporcionadas por la biblioteca de .NET, se creara un nuevo tipo de excepción.

Por ejemplo:

```
public class ValorNoValidoException : Exception
{
    public ValorNoValidoException()
    {
    }

    public ValorNoValidoException(string mensaje) : base(mensaje)
    {
```

```
    }

    //...
}
```

Esta nueva clase de excepción implementa dos constructores: uno sin parámetros y otro con un parámetro de tipo **string**; este parámetro es el mensaje de tipo **string** que devolverá la propiedad **Message** heredada de la clase **Exception**. El constructor *ValorNoValidoException* debe invocar al constructor de la clase base y pasar como argumento la cadena del mensaje, la que será almacenada en el miembro de datos correspondiente de la clase **Exception**.

Capitulo 4: LABORATORIO DE PROGRAMACION ORIENTADA A OBJETOS.

4.1 INTRODUCCION

Antes de entrar a la parte práctica se establece con los estudiantes el tiempo que disponen para programar las dos horas de laboratorio cada semana. Teniendo el horario en el que se realizaran las prácticas se procede a establecer el mecanismo de realización de las prácticas.

Se presentó en el capitulo 3 la planificación temporal para el contenido teórico y práctico de la asignatura Programación Orientada a Objetos.

Se dispone de 16 semanas para la realización de prácticas de laboratorio, además se plantearan ejercicios para que los estudiantes resuelvan después de que hayan terminado con las prácticas propuestas.

Se instalara el software necesario para trabajar el las practicas, eso lo realizará el docente previamente. El software a instalar es el SharpDevelop 2.2 y el .NET Framework SDK.

Se explicara el enunciado de las prácticas, se harán preguntas sobre ellas, con el objeto de indagar sobre los conocimientos que el alumno ha adquirido durante la clase teóricas.

4.1.1 Objetivos

- Que el estudiante aplique los conocimientos adquiridos en la clases teóricas sobre la Programación Orientada a Objetos

- Que el estudiante aprenda a resolver problemas orientados a objetos usando la herramienta SharpDevelop.

- Que el estudiante se formule problemas de la vida diaria y pueda desarrollarlos haciendo uso del ordenador.

4.1.2 Planificación Temporal.

La planificación del laboratorio es la siguiente:

PRACTICA	SEMANA	CONTENIDO	HORAS
1	1-2-3	Fundamentos de C#, Trabajando con Matrices.	6
2	4-5-6-7	Creación y definición de una Clase para el manejo de Horas.	8
3	8-9-10	Operadores Sobrecargados utilizando la clase CHora.	6
4	11-12-13-14	Derivación de la Clase **CEnfermo** y **CEmpleado** tomando como clase base la Clase **CFicha** de la Práctica 2. Implementación de la Clase **CHospital**.	8
5	15-16	Excepciones.	4
		TOTAL DE HORAS	**32**

PRÁCTICA 1: FUNDAMENTOS DE C#, TRABAJANDO CON MATRICES

OBJETIVO: Introducción a los fundamentos del lenguaje C#

4.1.3 Enunciado de la práctica.

Declarar la siguiente estructura para el manejo de matrices:

```
struct matriz
{
  int nFilas;        // Num. de filas
  int nColumnas;     // Num. de columnas
  int [,] Matrix;    // Datos de la matriz
};
```

Una variable de este tipo almacena en su campo **nFilas** el número de filas, en su campo **nColumnas** el número de columnas y en su campo **Matrix** la referencia a la matriz. Es decir, una variable de este tipo almacena toda la información necesaria para la creación y utilización de una matriz dinámica.

La aplicación deberá mostrar el siguiente menú:

1. Crear matriz 1
2. Crear matriz 2
3. Sumar las matrices
4. Multiplicar las matrices
5. Mostrar las matrices
6. Salir

4.2 PRÁCTICA 2: CREACIÓN Y DEFINICIÓN DE UNA CLASE PARA EL MANEJO DE HORAS.

OBJETIVO: Concepto de clase y objeto, atributos (datos miembro) y métodos (funciones miembro). Función externa.

4.2.1 Enunciado de la práctica.

Se debe crear en primer lugar una nueva clase Útil que contenga las funciones necesarias para leer los distintos tipos de datos y una función que permita imprimir un menú de opciones, el código es el siguiente:

```
//Clase util
using System;

namespace Ficha1
{
	public class Leer
	{
		public static short datoShort()
		{
			try
			{
				return Int16.Parse(Console.ReadLine());
			}
			catch(FormatException)
			{
				return Int16.MinValue; //valor más pequeño
			}
		}

		public static int datoInt()
		{
			try
			{
				return Int32.Parse(Console.ReadLine());
			}
			catch(FormatException)
			{
				return Int32.MinValue; //valor más pequeño
	}
		}

		public static long datoLong()
		{
			try
```

```
        {
            return Int64.Parse(Console.ReadLine());
        }
        catch(FormatException)
        {
            return Int64.MinValue; //valor más pequeño
        }
    }

    public static float datoFloat()
    {
        try
        {
            return Single.Parse(Console.ReadLine());
        }
        catch(FormatException)
        {
            return Single.NaN; // no es un numero; valor float
        }
    }

    public static double datoDouble()
    {
        try
        {
            return Double.Parse(Console.ReadLine());
        }
        catch(FormatException)
        {
            return Double.NaN; //no es un numero; valor double
        }
    }

    public static int Menu(string[] opciones, int numOpciones)
    {
        int i;
        int opcion = 0;
        Console.WriteLine("
_____________________________________________________");
        for (i = 1; i <= numOpciones; ++i)
            Console.WriteLine("          " + i + ".-" + opciones[i - 1]);
        Console.WriteLine("
_____________________________________________________");
        do
        {
            Console.WriteLine("(1 - " + numOpciones + "): ");
```

```
                        opcion = datoInt();
                    }
                    while (opcion < 1 || opcion > numOpciones);
                    return opcion;
                }

        }
}
```

A continuación se debe añadir al proyecto una nueva clase llamada CHora tal como se indica a continuación:

```
class  CHora
{
        private int Horas;
        private int Minutos;
        private int Segundos;
        private string formato; //Almacena "AM" , "PM" o "24"
        //Añadir  aquí una propiedad para acceder al formato

        public CHora()
        {       //Constructor sin parámetros
        }

        public CHora(int h, int m, int s, string f)
        {
                //Código del constructor con parámetros
        }

        public CHora(CHora hora)
        {
                //Código del constructor copia
        }

        public bool Format()
        {
                //Este método devuelve true si el formato está entre 12 h o falso si esta en
                formato 24 h
        }

        public bool VerificarHora()
        {
                //Devuelve true si la hora esta correcta, si no false. Llamar a la función
                //Format para averiguar en que formato esta la hora para luego hacer las
                comparaciones
        }
```

```
        public bool SetHora(int Horas, int nMinutos, int nSegundos, string format)
        {
                //Modifica el contenido del objeto
        }

        public void GetHora(ref int Horas, ref int nMinutos, ref int nSegundos, ref
        string format)
        {
                //Obtiene los datos del objeto
        }

        public string ToString()
        {
                //Devuelve una cadena con la hora (H:M:S FORMATO)
        }
}
```

Luego añadir una nueva clase al proyecto llamada **CFicha**, su funcionalidad es la siguiente:

```
class CFicha
{
        private string Nombre;
        private char Sexo;
        private Chora HoraNacimiento;
        public CFicha()
        {
                // Constructor sin parámetros
        }
        public CFicha (strinf nom, char sex, int h, int m, int s, string f)
        {
                // Constructor con parámetros
        }
        public CFicha (CFicha f)
        {
                // Constructor copia
        }
        public virtual CFicha clonar()
        {
                // Este método devuelve una copia del objeto que lo invocó.
                // Se declara virtual porque se redefinirá en las clases  derivadas de CFicha
        }
        public string GetNombre()
        {
                // Devuelve el nombre de la persona
```

```
        }
        public bool SetFicha (string nombre, char sexo, int h, int m, int s, string f)
        {
                // Modifica el contenido del objeto con los nuevos valores
        }
        public void GetFicha (ref string nombre, ref char sexo,  ref int h, ref  int m, ref
        int s, ref string f)
        {
                //Obtiene el contenido del objeto
        }
        public virtual void Mostrar()
        {
                // Imprime el contenido del objeto
        }
        public virtual string ToString()
        {
                // Devuelve en una cadena todo el contenido del objeto
        }
}
```

Desde la misma función Main poner a prueba las funciones definidas creando objetos de la clase CFicha e invocando a las distintas funciones.

4.3 PRÁCTICA 3: OPERADORES SOBRECARGADOS UTILIZANDO LA CLASE CHORA.

OBJETIVO: Concepto de sobrecarga de operador, operadores aritméticos sobrecargados, constructores con y sin parámetros, constructor copia.

4.3.1 Enunciado de la práctica.

Se añadirá a la clase implementada de CHora:

- Un constructor con todos los parámetros.
- Un constructor sin parámetros.
- Un constructor copia
- La sobrecarga del operador de resta.
- La sobrecarga del operador suma.

En el caso de los operadores de suma y resta debe tomar en cuenta lo siguiente:

- Solo pueden sumar o restar horas que tengan el mismo formato.
- En el caso de la suma, la hora resultante debe ser correcta. Esto es, si la suma de los segundos es mayor que 60, se debe restar 60 y aumentar en 1 a los minutos. Si los minutos son mayores que 60, se debe restar 60 y aumentar en 1 a las horas. Dependiendo del formato de la hora, se debe restar 12 o 24 si esta excede ese valor.
- En el caso de la resta, la hora resultante debe ser correcta. Esto es, si la resta de los segundos es menor que 60, se debe sumar 60 y disminuir en 1 a los minutos. Si los minutos son menores que 60, se debe sumar 60 y disminuir en 1 a las horas. Dependiendo del formato de la hora, se debe sumar 12 o 24 si esta es menor que ese valor.

4.4 PRÁCTICA 4: DERIVACIÓN DE LA CLASE CENFERMO Y CEMPLEADO TOMANDO COMO CLASE BASE LA CLASE CFICHA DE LA PRÁCTICA 2. IMPLEMENTACIÓN DE LA CLASE CHOSPITAL.

OBJETIVO: Conceptos de herencia y polimorfismo.

4.4.1 Enunciado de la práctica.

La clase **CEmpleado** tendrá, además de la funcionalidad heredada de la clase **CFicha**, los siguientes datos miembro privados:

```
string m_sCategoria;   // (ej.: "Enfermera", "Médico"...)
int m_nAntiguedad;     // (ej.: 3)
```

NOTA: Se deben añadir a esta clase el constructor sin argumentos, constructor con argumentos y constructor copia. Recuerde que se deben invocar a los correspondientes constructores de la clase base.

La siguiente función miembro servirá para modificar el contenido de los objetos de la clase:

```
public bool SetEmpleado(string categoria, int antiguedad, string nombre,
char sexo, int h, int m, int s, string f)
```

La siguiente función miembro servirá para obtener el contenido de los objetos de la clase:

```
public void GetEmpleado(out string categoria, out int antiguedad, out string
nombre, out char sexo, out int h, out int m, out int s, out string f)
```

NOTA: Se deben redefinir los métodos Mostrar, ToString y clonar.

La clase **CEnfermo** (que también será derivada de **CFicha**) solo tendrá un dato miembro:

```
string m_sEnfermedad;     // (ej.: "Bronquitis"...)
```

y las correspondientes funciones miembro **Set...** y **Get...**:

```
public bool SetEnfermo(string enfermedad, string nombre, char sexo, int h,
int m, int s, string f)

public void GetEnfermo(out string enfermedad, out string nombre, out char
sexo, out int h, out int m, out int s, out string f)
```

El funcionamiento será análogo al de la clase **CEmpleado**.

Se declarará una nueva clase **CHospital** mediante la cuál se implementará un array de objetos de tipo empleado o enfermo. Habrá que declarar y definir:

- Un constructor que crea el array de objetos. Recibirá como parámetro el número máximo de elementos del array.
- El constructor copia. Observe que la clase **CHospital** contiene referencias a objetos. Eso significa que el el constructor copia por omisión sólo copia direcciones, no duplicarán los objetos referenciados.
- Una función **SetPersona()**, que permita almacenar en el array una referencia a los datos de cada persona, ya sea empleado o enfermo.
- Una función **MostrarGente()**, que llama a la función **Mostrar()** para cada uno de los empleados o enfermos del array.

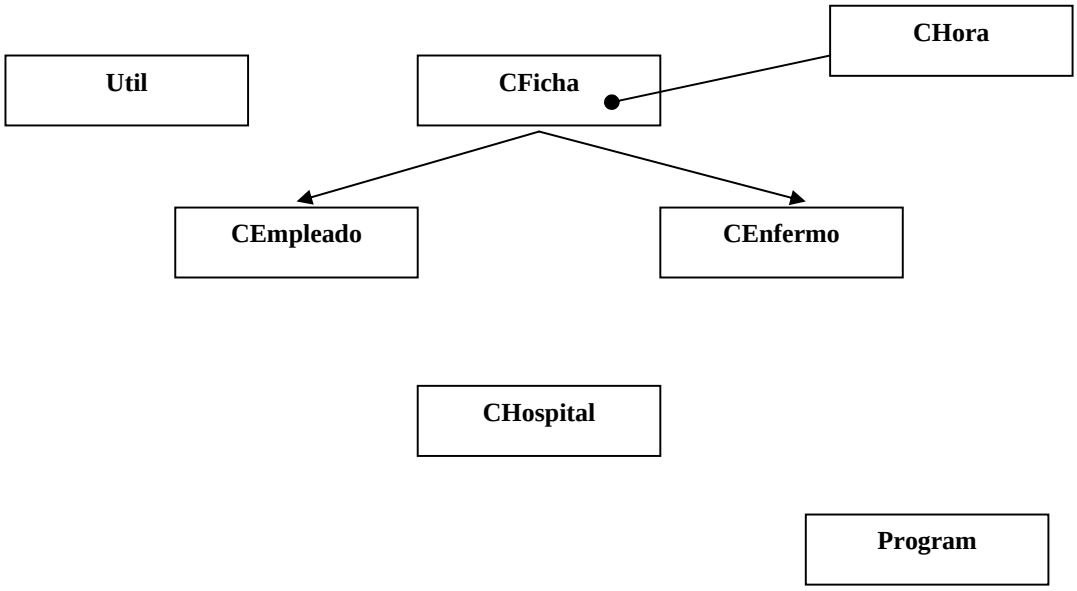
El programa principal se contendrá un menú como el siguiente:

1. Introducir empleado.
2. Introducir enfermo.
3. Buscar por nombre.
4. Mostrar hospital.
5. Copia de seguridad del hospital.
6. Restaurar copia de seguridad.
7. Salir.

La opción 4 mostrará todas las personas del hospital, empleados y enfermos (hacer una pausa cuando se llene la pantalla o por cada persona mostrada).

La opción 5 hará un duplicado del objeto hospital actual. Una vez hecha la copia de seguridad suponer que, haciendo pruebas, se altera la composición del hospital (por ejemplo, añadiendo nuevos enfermos) y se desea volver a la composición que había hecho cuando se hizo la copia de seguridad. La opción 6 permitirá esta última acción. Se puede comprobar con la opción 4.

Realizar una segunda versión sustituyendo el array de referencias a objetos por una lista (clase genérica **List**) definido así: **List<CFicha > Gente**.


CHora
Util
CFicha
CEmpleado
CEnfermo
CHospital
Program

4.5 PRÁCTICA 5: EXCEPCIONES.

OBJETIVO: Concepto de excepciones, lanzamiento y captura de excepciones.

4.5.1 Enunciado de la práctica.

Construir una aplicación que cumpla con los requisitos siguientes:

1. Declarar una clase para el manejo de objetos de tipo **Temperatura**. Los datos son el **valor** de la temperatura y la **hora** en la que fue registrada. Este último dato es un objeto de la clase CHora implementada en la práctica 2.

2. Declarar una clase llamada **ExcTemperatura** para manejar excepciones relacionadas con la manipulación de objetos de tipo Temperatura. Esta clase tiene un único método llamado **Mensaje** que da información del error.

3. La clase Temperatura tiene un método que lee los datos. Se debe lanzar una excepción de tipo **ExcTemperatura** si el valor de la temperatura no se encuentra en el rango entre 0 y 100. También se deben manejar las excepciones asociadas a la incorrecta lectura de un tipo de datos.

4. En el programa principal, poner a prueba el lanzamiento y la captura de las excepciones asociadas con la manipulación de objetos de tipo Temperatura.

BIBLIOGRAFÍA

- Ceballos Sierra, Francisco Javier. Microsoft C# Lenguaje y aplicaciones. Editorial RA-MA
- Ceballos Sierra, Francisco Javier. Microsoft C#, Curso de Programación. RA-MA
- http://www.devjoker.com/contenidos/Tutorial-C/141/Espacio-de-nombres-distribuidos.aspx
- http://es.wikipedia.org/wiki/.NET_de_Microsoft
- http://geneura.ugr.es/~jmerelo/ws/
- http://support.microsoft.com/kb/305140/es
- http://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos
- http://www.monografias.com/trabajos14/progorie/progorie.shtml

ANEXOS

Solución de la práctica 1

```
/*CMatriz.cs*/
using System;
namespace ProblemaMatriz
{
    class Matriz
    {
        private int[,] datos;


        public Matriz(int n)
        {
            datos = new int[n, n]; // Se crea la matriz del tamaño
indicado
        }

        public void Cargar()
        {
            int i, j;

            Console.WriteLine("\nIngrese los Valores para la matriz:");
            for (i = 0; i < datos.GetLength(0); i++)
            {
                Console.WriteLine("Fila {0}:", i);
                for (j = 0; j < datos.GetLength(1); j++)
                {
                    Console.Write("\tCelda {0}: ", j);
                    datos[i, j] = Util.datoInt();
                }
            }
        }

        int Tamano()
        {
            return (datos.GetLength(0));
        }

        //Metodo que suma dos matrices
        public static Matriz Sumar(Matriz A, Matriz B)
        {
            Matriz m = new Matriz(A.Tamano());

            //El siguiente ciclo con anidamiento de 3 nivel realiza la
multiplicación
            for (int i = 0; i < m.Tamano(); i++)
            {
                for (int j = 0; j < m.Tamano(); j++)
                {
                    m.datos[i, j] = (A.datos[i, j] + B.datos[i, j]);
                }
            }
            return (m);
        }

        //Metodo que multiplica dos matrices
```

```
        public static Matriz Multiplicar(Matriz A, Matriz B)
        {
            Matriz m = new Matriz(A.Tamano());

            //El siguiente ciclo con anidamiento de 3 nivel realiza la
multiplicación
            for (int i = 0; i < m.Tamano(); i++)
            {
                for (int j = 0; j < m.Tamano(); j++)
                {
                    m.datos[i, j] = 0;
                    for (int k = 0; k < m.Tamano(); k++)
                        m.datos[i, j] += (A.datos[i, k] * B.datos[k,
j]);
                }
            }
            return (m);
        }

        // Mostrar(): muestra en pantalla (consola) la matriz.
        public void Mostrar()
        {
            Console.WriteLine("\n");
            // Se imprime por filas completas hacia el lado
            for (int i = 0; i < datos.GetLength(0); i++)
            {
                Console.Write("| "); // Se imprime un separador
                for (int j = 0; j < datos.GetLength(1); j++)
                    Console.Write("{0} ", datos[i, j]);
                Console.WriteLine("|"); // Se imprime un separador
            }
            // Al final de recorrer la fila, se cambia de línea
            Console.WriteLine();
        }
    }
}
```

```
/*Util.cs*/
using System;
namespace ProblemaMatriz
{
    public class Util    // clase Leer
    {
            public static short datoShort()
            {
                try
                {
                    return Int16.Parse(Console.ReadLine());
                }
                catch (FormatException)
                {
                    return Int16.MinValue; // valor más pequeño
                }
            }

            public static int datoInt()
            {
```

```
                try
                {
                    return Int32.Parse(Console.ReadLine());
                }
                catch (FormatException)
                {
                    return Int32.MinValue; // valor más pequeño
                }
            }

            public static long datoLong()
            {
                try
                {
                    return Int64.Parse(Console.ReadLine());
                }
                catch (FormatException)
                {
                    return Int64.MinValue; // valor más pequeño
                }
            }

            public static float datoFloat()
            {
                try
                {
                    return Single.Parse(Console.ReadLine());
                }
                catch (FormatException)
                {
                    return Single.NaN; // No es un Número; valor float.
                }
            }

            public static double datoDouble()
            {
                try
                {
                    return Double.Parse(Console.ReadLine());
                }
                catch (FormatException)
                {
                    return Double.NaN; // No es un Número; valor
double.
                }
            }

            public static int Menu(string[] opciones, int numOpciones)
            {
                int i;
                int opcion = 0;


System.Console.WriteLine("\n_____________________________________________\
n\n");
                for (i = 1; i <= numOpciones; ++i)
```

```
                    System.Console.WriteLine("    " + i + ".- " +
opciones[i - 1]);

System.Console.WriteLine("_____________________________________________\n"
);

                do
                {
                    System.Console.WriteLine("\nOpción (1 - " +
numOpciones + "): ");
                    opcion = datoInt();
                }
                while (opcion < 1 || opcion > numOpciones);

                return opcion;
            }
        }
}
```

```
/*Test.cs*/
using System;
namespace ProblemaMatriz
{
    class Test
    {
        public static void Main(string[] args)
        {
            int N; // Dimension de las matrices, dada por el usuario
            Console.Write("Ingrese el valor de la dimension de las
matrices (N) de 1 a 10: ");
            N = Util.datoInt();
            Matriz A = new Matriz(N); // Se declaran e inicializan las
dos matrices A y B
            Matriz B = new Matriz(N);
            int opcion;
            string[] opciones =
            {
              "Crear matriz 1",
              "Crear matriz 2",
              "Sumar las matrices",
              "Multiplicar las matrices",
              "Mostrar las matrices",
              "Salir"
            };

            do
            {
                opcion = Util.Menu(opciones, opciones.Length);
                switch (opcion)
                {
                    case 1:
                        A.Cargar();
                        break;
                    case 2:
                        B.Cargar();
                        break;
                    case 3:
```

```
                        Console.WriteLine("La suma de las matrices
es:");
                        Matriz X = Matriz.Sumar(A, B);
                        X.Mostrar();
                        break;
                    case 4:
                        Console.WriteLine("La multiplicacion de las
matrices es:");
                        Matriz C = Matriz.Multiplicar(A, B);
                        C.Mostrar();
                        break;
                    case 5:
                        A.Mostrar();
                        B.Mostrar();
                        break;
                    case 6:
                        Console.Write("Presione ENTER para
terminar...");
                        break;
                }
            } while (opcion != 6);
            Console.ReadLine();
        }
    }
}
```

Solución de la práctica 2

```
/*CHora.cs*/
using System;

namespace CHora
{
    class CHora
    {
        private int Horas;
        private int Minutos;
        private int Segundos;
        private string formato;

        public CHora()
        {
            Horas = 0;
            Minutos = 0;
            Segundos = 0;
            formato = "";
        }

        public CHora(int h, int m, int s, string f)
        {
            Horas = h;
            Minutos = m;
            Segundos = s;
            formato = f;
        }
```

```
        public CHora(CHora hora)
        {
            Horas = hora.Horas;
            Minutos = hora.Minutos;
            Segundos = hora.Segundos;
            formato = hora.formato;
            System.Console.WriteLine("Constructor copia de CHora
invocado...");
        }

        public bool Formato()
        {
            if (formato == "AM" || formato == "PM")
                return true;
            else
                return false;
        }

        public bool VerificarHora()
        {
            bool f = false, h = false, m = false, s = false;
            System.Console.WriteLine("Verificando la hora...");
            f = Formato();
            switch (f)
            {
                case false:
                    h = ((Horas >= 0) && (Horas <= 23));
                    m = ((Minutos >= 0) && (Minutos <= 59));
                    s = ((Segundos >= 0) && (Segundos <= 59));
                    break;
                case true:
                    h = ((Horas >= 0) && (Horas <= 11));
                    m = ((Minutos >= 0) && (Minutos <= 59));
                    s = ((Segundos >= 0) && (Segundos <= 59));
                    break;
            }//fin switch

            if (h && m && s)
            {
                System.Console.WriteLine("La Hora esta CORRECTA...");
                return (true);
            }
            else
            {
                System.Console.WriteLine("La Hora esta INCORRECTA...");
                return (false);
            }
        }

        public bool SetHora(int nHoras, int nMinutos, int nSegundos,
string format)
        {
            Horas = nHoras;
            Minutos = nMinutos;
            Segundos = nSegundos;
            formato = format;
            return VerificarHora();
```

```
        }

        public void GetHora()
        {
            System.Console.WriteLine(Horas + ":" + Minutos + ":" +
Segundos + " " + formato);
        }
    }
}
```

```
/* CFicha.cs*/
using System;

namespace CHora
{
    class CFicha
    {
        private string Nombre;
        private char Sexo;
        private CHora HoraNacimiento;

        public CFicha()
        {
            Nombre = "";
            Sexo = '\a';
            HoraNacimiento = new CHora();
        }

        public CFicha(string nom, char sex, int h, int m, int s, string
f)
        {
            Nombre = nom;
            Sexo = sex;
            HoraNacimiento = new CHora(h, m, s, f);
            System.Console.WriteLine("Constructor de CFicha con
argumentos invocado...");
        }

        public CFicha(CFicha f)
        {
            Nombre = f.Nombre;
            Sexo = f.Sexo;
            HoraNacimiento = new CHora(f.HoraNacimiento);
            System.Console.WriteLine("Constructor copia de CFicha
invocado...");
        }

        public virtual CFicha clonar()
        {
            return new CFicha(this);
        }

        public string GetNombre()
        {
            return Nombre;
        }
```

```
        public void SetFicha()
        {
            int h, m, s;
            string f;
            bool aux = false;

            System.Console.WriteLine("Introduzca el nombre: ");
            Nombre = System.Console.ReadLine();
            System.Console.WriteLine("Introduzca el sexo: ");
            Sexo = (char)System.Console.Read();
            System.Console.ReadLine(); //limpiar el buffer de entrada

            do
            {
                System.Console.WriteLine("Introduzca la hora: ");
                h = Leer.datoInt();
                System.Console.WriteLine("Introduzca los minutos: ");
                m = Leer.datoInt();
                System.Console.WriteLine("Introduzca los segundos: ");
                s = Leer.datoInt();
                do
                {
                    System.Console.WriteLine("Introduzca el formato =>
AM, PM o 24 ");
                    f = System.Console.ReadLine();
                    if (f == "AM" || f == "PM" || f == "24")
                        aux = true;  //Correcto
                    else
                        aux = false;
                    if (aux == false)
                        System.Console.WriteLine("El formato
introducido es incorrecto. ");
                } while (aux == false);

                aux = HoraNacimiento.SetHora(h, m, s, f);

            } while (aux == false);
        }

        public virtual void Mostrar()
        {
            System.Console.WriteLine("Nombre: " + Nombre);
            System.Console.WriteLine("Sexo: " + Sexo);
            HoraNacimiento.GetHora();
        }
    }
}
```

```
/*Leer.cs*/
using System;

namespace CHora
{
    public class Leer
    {
        public static short datoShort()
```

```
        {
            try
            {
                return Int16.Parse(Console.ReadLine());
            }
            catch (FormatException)
            {
                return Int16.MinValue; //valor mas pequeno
            }
        }

        public static int datoInt()
        {
            try
            {
                return Int32.Parse(Console.ReadLine());
            }
            catch (FormatException)
            {
                return Int32.MinValue; //valor mas pequeno
            }
        }

        public static long datoLong()
        {
            try
            {
                return Int64.Parse(Console.ReadLine());
            }
            catch (FormatException)
            {
                return Int64.MinValue; //valor mas pequeno
            }
        }

        public static float datoFloat()
        {
            try
            {
                return Single.Parse(Console.ReadLine());
            }
            catch (FormatException)
            {
                return Single.NaN; // no es un numero; valor float
            }
        }

        public static double datoDouble()
        {
            try
            {
                return Double.Parse(Console.ReadLine());
            }
            catch (FormatException)
            {
                return Double.NaN; //no es un numero; valor double
            }
```

```
        }

        public static int Menu(string[] opciones, int numOpciones)
        {
            int i;
            int opcion = 0;
            Console.WriteLine(" \n
_____________________________________________________\n\n");
            for (i = 1; i <= numOpciones; ++i)
                Console.WriteLine("       " + i + ".-" + opciones[i -
1]);
            Console.WriteLine("
_____________________________________________________\n");
            do
            {
                Console.WriteLine("\nOpcion (1 - " + numOpciones + "):
");
                opcion = datoInt();
            }
            while (opcion < 1 || opcion > numOpciones);
            return opcion;
        }
    }
}
```

```
/* Test.cs*/
using System;

namespace CHora
{
    public class Test
    {

        public static void Main(string[] args)
        {
            CFicha ficha = new CFicha();

            int opcion;

            string [] opciones =
            {
              "Llenar ficha",
              "Mostrar ficha",
              "Salir"
            };

            do
            {
                opcion = Leer.Menu(opciones, opciones.Length);
                switch (opcion)
                {
                    case 1:
                        ficha.SetFicha();
                        break;
                    case 2:
                        ficha.Mostrar();
```

```
                        break;
                    case 3:
                        break;
                }
            } while (opcion != 3);
        }
    }
}
```

Solución de la práctica 3

```
/*CHora.cs*/
using System;

namespace CHora
{
      public class CHora
      {
            //atributos
            private int Horas;
            private int Minutos;
            private int Segundos;
            private string formato;  // Almacena "AM", "PM" o "24"

            //metodos
            protected bool Format()
            {
                  return
((System.String.Compare("AM",formato.ToUpper(),
true)==0)||((System.String.Compare("PM",formato.ToUpper(), true)==0)));
            }

            public void AsignarHora(int hh, int mm, int ss, string ff)
            {
                  Horas = hh;
                  Minutos = mm;
                  Segundos = ss;
                  formato = ff;
            }

            public void ObtenerHora(out int hh, out int mm, out int ss,
out string ff)
            {
                  hh = Horas;
                  mm = Minutos;
                  ss = Segundos;
                  ff = formato;
            }

            public CHora(int h, int m, int s, string f)
            {
                  Horas = h;
                  Minutos = m;
                  Segundos = s;
                  formato = f;
            }
```

```
            public CHora()
            {
            }

            public CHora(CHora hora)
            {
                  Horas = hora.Horas;
                  Minutos = hora.Minutos;
                  Segundos = hora.Segundos;
                  formato = hora.formato;
            }

            // suma de horas
            public static CHora operator +(CHora h1, CHora h2)
            {
                  CHora suma = new CHora();
                  if (h1.formato == h2.formato)
                  {
                        suma = new CHora(h1.Horas + h2.Horas,h1.Minutos
+ h2.Minutos, h1.Segundos + h2.Segundos,h1.formato);
                        if (suma.Segundos > 60)
                        {
                              suma.Segundos -= 60;
                              suma.Minutos++;
                        }
                        if (suma.Minutos > 60)
                        {
                              suma.Minutos -= 60;
                              suma.Horas++;
                        }
                    return suma;
                  }
                  else
                  {
                        Console.WriteLine("Las horas no se pueden
sumar, tienen diferente formato");
                        return suma;
                  }
            }

            // resta de horas
            public static CHora operator -(CHora h1, CHora h2)
            {
                  CHora resta = new CHora();
                  if (h1.formato == h2.formato)
                  {
                        resta = new CHora(h1.Horas -
h2.Horas,h1.Minutos - h2.Minutos, h1.Segundos -
h2.Segundos,h1.formato);
                    if (resta.Segundos < 0)
                    {
                        resta.Segundos += 60;
                        resta.Minutos--;
                    }
                    if (resta.Minutos < 0)
                    {
```

```
                    resta.Minutos += 60;
                    resta.Horas--;
                }
                        return resta;
                }
                else
                {
                        Console.WriteLine("Las horas no se pueden
restar, tienen diferente formato");
                        return resta;
                }
            }

            //mostrar suma de horas
            public override string ToString()
            {
                  return (String.Format("{0}:" + "{1}:" + "{2} " +
"{3}",Horas, Minutos, Segundos, formato));
            }

            public bool EsHoraCorrecta()
            {
                  bool  h, m, s;
                  Console.WriteLine("Verificando la hora...");
                  if (Format())
                  {
                        h = ((Horas>0) && (Horas <= 12));
                        m = ((Minutos>=0) && (Minutos <=59));
                        s = ((Segundos>=0)&& (Segundos <= 59));
                  }
                  else
                  {
                        h = ((Horas>= 0) && (Horas <= 23));
                        m = ((Minutos>=0) && (Minutos <=59));
                        s = ((Segundos>=0)&& (Segundos <= 59));
                  }
                  return h && m && s;
            }
      }
}
```

```
/*Leer.cs*/
using System;

namespace CHora
{
    public class Leer
    {
        public static short datoShort()
        {
            try
            {
                return Int16.Parse(Console.ReadLine());
            }
            catch (FormatException)
            {
```

```
                return Int16.MinValue; //valor mas pequeno
            }
        }

        public static int datoInt()
        {
            try
            {
                return Int32.Parse(Console.ReadLine());
            }
            catch (FormatException)
            {
                return Int32.MinValue; //valor mas pequeno
            }
        }

        public static long datoLong()
        {
            try
            {
                return Int64.Parse(Console.ReadLine());
            }
            catch (FormatException)
            {
                return Int64.MinValue; //valor mas pequeno
            }
        }

        public static float datoFloat()
        {
            try
            {
                return Single.Parse(Console.ReadLine());
            }
            catch (FormatException)
            {
                return Single.NaN; // no es un numero; valor float
            }
        }

        public static double datoDouble()
        {
            try
            {
                return Double.Parse(Console.ReadLine());
            }
            catch (FormatException)
            {
                return Double.NaN; //no es un numero; valor double
            }
        }
    }
}
```

```
/*Test.cs*/
using System;
```

```
namespace CHora
{
    public class Test
    {
        //Leer una Hora
        public static void LeerHora(ref int h, ref int m, ref int s,
ref string f)
        {
            bool aux = false;
            Console.Write("Introduzca la Hora: ");
            h = Leer.datoInt();
            Console.Write("Introduzca los Minutos: ");
            m = Leer.datoInt();
            Console.Write("Introduzca los Segundos: ");
            s = Leer.datoInt();
            do
            {
                Console.Write("Introduzca el formato ==> AM, PM o 24:
");
                f = Console.ReadLine();
                if ((String.Compare("AM", f, true) == 0) ||
(String.Compare("PM", f, true) == 0) || (String.Compare("24", f, true)
== 0))
                    aux = true;
                else
                    aux = false;
                if (aux == false)
                    Console.WriteLine("El formato introducido es
incorrecto. ");
            } while (aux == false);
        }
        //Visualizar una hora
        public static void VisualizarHora(CHora hora)
        {
            int h, m, s; string f;
            hora.ObtenerHora(out h, out m, out s, out f);
            Console.WriteLine("La hora es ... ");
            Console.WriteLine(h + ":" + m + ":" + s + " " + f);
        }

        //Establecer una hora, verificarla y visualizarla
        public static void Main(string[] args)
        {
            CHora hora = new CHora();
            CHora hora1 = new CHora();
            CHora d, m;
            int Horas = 0, Minutos = 0, Segundos = 0;
            string formato = " ";
            do
            {
                LeerHora(ref Horas, ref Minutos, ref Segundos, ref
formato);
                hora.AsignarHora(Horas, Minutos, Segundos, formato);
            }
            while (!hora.EsHoraCorrecta());
            VisualizarHora(hora);
            Console.WriteLine();
```

```
            Console.WriteLine("Introduzca otra hora: ");
            do
            {
                LeerHora(ref Horas, ref Minutos, ref Segundos, ref
formato);
                hora1.AsignarHora(Horas, Minutos, Segundos, formato);
            }
            while (!hora1.EsHoraCorrecta());
            VisualizarHora(hora1);
            d = hora + hora1;
            Console.WriteLine("La suma de las horas es: ");
            Console.WriteLine(d.ToString());
            CHora h3 = new CHora(hora1); //inicia un objeto asignandole
            m = hora - hora1;
            Console.WriteLine("La resta de las horas es: ");
            Console.WriteLine(m.ToString());
            Console.ReadLine();
        }
    }
}
```

Solución de la práctica 4

```
/* CFicha.cs*/
using System;

namespace Hospital
{
    class CFicha
    {
        private string Nombre;
        private char Sexo;
        private CHora HoraNacimiento;

        public CFicha()
        {
            Nombre = "";
            Sexo = '\a';
            HoraNacimiento = new CHora();
            System.Console.WriteLine("Constructor de CFicha
invocado...");
        }

        public CFicha(string nom, char sex, int h, int m, int s, string
f)
        {
            Nombre = nom;
            Sexo = sex;
            HoraNacimiento = new CHora(h, m, s, f);
            System.Console.WriteLine("Constructor de CFicha con
argumentos invocado...");
        }

        public CFicha(CFicha f)
        {
            Nombre = f.Nombre;
```

```
            Sexo = f.Sexo;
            HoraNacimiento = new CHora(f.HoraNacimiento);
            System.Console.WriteLine("Constructor copia de CFicha
invocado...");
        }

        public virtual CFicha clonar()
        {
            return new CFicha(this);
        }

        public string GetNombre()
        {
            return Nombre;
        }

        public void SetFicha()
        {
            int h, m, s;
            string f;
            bool aux = false;

            System.Console.WriteLine("Introduzca el nombre: ");
            Nombre = System.Console.ReadLine();
            System.Console.WriteLine("Introduzca el sexo: ");
            Sexo = (char)System.Console.Read();
            System.Console.ReadLine(); //limpiar el buffer de entrada

            do
            {
                System.Console.WriteLine("Introduzca la hora: ");
                h = Util.datoInt();
                System.Console.WriteLine("Introduzca los minutos: ");
                m = Util.datoInt();
                System.Console.WriteLine("Introduzca los segundos: ");
                s = Util.datoInt();
                do
                {
                    System.Console.WriteLine("Introduzca el formato =>
AM, PM o 24 ");
                    f = System.Console.ReadLine();
                    if (f == "AM" || f == "PM" || f == "24")
                        aux = true;  //Correcto
                    else
                        aux = false;
                    if (aux == false)
                        System.Console.WriteLine("El formato
introducido es incorrecto. ");
                } while (aux == false);

                aux = HoraNacimiento.SetHora(h, m, s, f);

            } while (aux == false);
        }

        public virtual void Mostrar()
        {
```

```
            System.Console.WriteLine("Nombre: " + Nombre);
            System.Console.WriteLine("Sexo: " + Sexo);
            HoraNacimiento.GetHora();
        }
    }
}
```

```
/*CEmpleado.cs*/
using System;

namespace Hospital
{
    class CEmpleado : CFicha
    {
        private string Categoria;
        private int Antiguedad;

        public CEmpleado()
            : base()
        {
            Categoria = "";
            Antiguedad = 0;
            System.Console.WriteLine("Constructor de CEmpleado
invocado...");
        }

        public CEmpleado(string nom, char sex, int h, int m, int s,
string f, string c, int a)
            : base(nom, sex, h, m, s, f)
        {
            Categoria = c;
            Antiguedad = a;
            System.Console.WriteLine("Constructor de CEmpleado con
argumentos invocado...");
        }

        public CEmpleado(CEmpleado emp)
            : base(emp)
        {
            Categoria = emp.Categoria;
            Antiguedad = emp.Antiguedad;
            System.Console.WriteLine("Constructor copia de CEmpleado
invocado...");
        }

        public override CFicha clonar()
        {
            return new CEmpleado(this);
        }

        public void SetEmpleado()
        {
            base.SetFicha();
            System.Console.WriteLine("Introduzca la categoria: ");
            Categoria = System.Console.ReadLine();
            System.Console.WriteLine("Introduzca la antiguedad: ");
            Antiguedad = Util.datoInt();
```

```
        }

        public override void Mostrar()
        {
            base.Mostrar();
            System.Console.WriteLine("Categoria: " + Categoria);
            System.Console.WriteLine("Antiguedad: " + Antiguedad);
            System.Console.WriteLine("----------------------------------
------------");
        }
    }
}
```

```
/*CEnfermo.cs*/
using System;

namespace Hospital
{
    class CEnfermo : CFicha
    {
        private string Enfermedad;

        public CEnfermo()
            : base()
        {
            Enfermedad = "";
            System.Console.WriteLine("Constructor de CEnfermo
invocado...");
        }

        public CEnfermo(string nom, char sex, int h, int m, int s,
string f, string e)
            : base(nom, sex, h, m, s, f)
        {
            Enfermedad = e;
            System.Console.WriteLine("Constructor de CEnfermo con
argumentos invocado...");
        }

        public CEnfermo(CEnfermo enf)
            : base(enf)
        {
            Enfermedad = enf.Enfermedad;
            System.Console.WriteLine("Constructor copia de CEnfermo
invocado...");
        }

        public override CFicha clonar()
        {
            return new CEnfermo(this);
        }

        public void SetEnfermo()
        {
            base.SetFicha();
            System.Console.WriteLine("Introduzca la enfermedad: ");
            Enfermedad = System.Console.ReadLine();
```

```
        }

        public override void Mostrar()
        {
            base.Mostrar();
            System.Console.WriteLine("Enfermedad: " + Enfermedad);
            System.Console.WriteLine("----------------------------------
------------");
        }
    }
}
```

```
/*CHora.cs*/
using System;

namespace Hospital
{
    class CHora
    {
        private int Horas;
        private int Minutos;
        private int Segundos;
        private string formato;

        public CHora()
        {
            Horas = 0;
            Minutos = 0;
            Segundos = 0;
            formato = "";
            System.Console.WriteLine("Constructor de CHora
invocado...");
        }

        public CHora(int h, int m, int s, string f)
        {
            Horas = h;
            Minutos = m;
            Segundos = s;
            formato = f;
            System.Console.WriteLine("Constructor de CHora con
argumentos invocado...");
        }

        public CHora(CHora hora)
        {
            Horas = hora.Horas;
            Minutos = hora.Minutos;
            Segundos = hora.Segundos;
            formato = hora.formato;
            System.Console.WriteLine("Constructor copia de CHora
invocado...");
        }

        public bool Formato()
        {
```

```
            if (formato == "AM" || formato == "PM")
                return true;
            else
                return false;
        }

        public bool VerificarHora()
        {
            bool f = false, h = false, m = false, s = false;
            System.Console.WriteLine("Verificando la hora...");
            f = Formato();
            switch (f)
            {
                case false:
                    h = ((Horas >= 0) && (Horas <= 23));
                    m = ((Minutos >= 0) && (Minutos <= 59));
                    s = ((Segundos >= 0) && (Segundos <= 59));
                    break;
                case true:
                    h = ((Horas >= 0) && (Horas <= 11));
                    m = ((Minutos >= 0) && (Minutos <= 59));
                    s = ((Segundos >= 0) && (Segundos <= 59));
                    break;
            }//fin switch

            if (h && m && s)
            {
                System.Console.WriteLine("La Hora esta CORRECTA...");
                return (true);
            }
            else
            {
                System.Console.WriteLine("La Hora esta INCORRECTA...");
                return (false);
            }
        }

        public bool SetHora(int nHoras, int nMinutos, int nSegundos,
string format)
        {
            Horas = nHoras;
            Minutos = nMinutos;
            Segundos = nSegundos;
            formato = format;
            return VerificarHora();
        }

        public void GetHora()
        {
            System.Console.WriteLine(Horas + ":" + Minutos + ":" +
Segundos + " " + formato);
        }
    }
}
```

```
/*CHospital.cs*/
```

```
using System;

namespace Hospital
{
    class CHospital
    {
        private CFicha[] Gente;
        private int Maximo;
        private int NumeroObjetos;

        public CHospital(int n)
        {
            NumeroObjetos = 0;
            Maximo = n;
            Gente = new CFicha[Maximo];
        }

        public CHospital(CHospital h)
        {
            NumeroObjetos = h.NumeroObjetos;
            Maximo = h.Maximo;
            Gente = new CFicha[Maximo];
            for (int j = 0; j < NumeroObjetos; j++)
                Gente[j] = h.Gente[j].clonar();
            System.Console.WriteLine("Constructor Copia de CHospital
invocado. \n");
        }

        public void SetPersona(CFicha f)
        {
            if (NumeroObjetos < Maximo)
                Gente[NumeroObjetos++] = f.clonar(); //OJO:
Polimorfismo
        }

        public void MostrarGente()
        {
            for (int i = 0; i < NumeroObjetos; i++)
                Gente[i].Mostrar();
        }

        public void BuscarPersona(string persona)
        {
            bool encontrado = false;
            for (int i = 0; i < NumeroObjetos; i++)
            {
                if (Gente[i].GetNombre() == persona)
                {
                    System.Console.WriteLine("Se ha encontrado... " +
persona);
                    System.Console.WriteLine("----------------------
--------------------");
                    Gente[i].Mostrar();
                    encontrado = true;
                }
            }
            if (!encontrado)
```

```
                System.Console.WriteLine("No se ha encontrado... " +
persona);
            }
        }
}
```

```
/*CUtil.cs*/
using System;

namespace Hospital // espacio de nombres
{
    public class Util    // clase Leer
    {
        public static short datoShort()
        {
            try
            {
                return Int16.Parse(Console.ReadLine());
            }
            catch (FormatException)
            {
                return Int16.MinValue; // valor más pequeño
            }
        }

        public static int datoInt()
        {
            try
            {
                return Int32.Parse(Console.ReadLine());
            }
            catch (FormatException)
            {
                return Int32.MinValue; // valor más pequeño
            }
        }

        public static long datoLong()
        {
            try
            {
                return Int64.Parse(Console.ReadLine());
            }
            catch (FormatException)
            {
                return Int64.MinValue; // valor más pequeño
            }
        }

        public static float datoFloat()
        {
            try
            {
                return Single.Parse(Console.ReadLine());
            }
            catch (FormatException)
```

```
            {
                return Single.NaN; // No es un Número; valor float.
            }
        }

        public static double datoDouble()
        {
            try
            {
                return Double.Parse(Console.ReadLine());
            }
            catch (FormatException)
            {
                return Double.NaN; // No es un Número; valor double.
            }
        }

        public static int Menu(string[] opciones, int numOpciones)
        {
            int i;
            int opcion = 0;


System.Console.WriteLine("\n_____________________________________________\
n\n");
            for (i = 1; i <= numOpciones; ++i)
                System.Console.WriteLine("    " + i + ".- " +
opciones[i - 1]);

System.Console.WriteLine("_____________________________________________\n"
);

            do
            {
                System.Console.WriteLine("\nOpción (1 - " + numOpciones
+ "): ");
                opcion = datoInt();
            }
            while (opcion < 1 || opcion > numOpciones);

            return opcion;
        }
    }
}
```

```
/*Test.cs*/
using System;

namespace Hospital
{
    class Program
    {
        static void Main(string[] args)
        {

            int opcion;
```

```
            string nom;

            CEmpleado emp = new CEmpleado();
            CEnfermo enf = new CEnfermo();

            CHospital hospital = new CHospital(100);
            CHospital hospital_copia = new CHospital(100);

            string[] opciones =
            {
              "Introducir empleado",
              "Introducir enfermo",
              "Buscar por nombre",
              "Mostrar hospital",
              "Copia de seguridad del hospital",
              "Restaurar copia de seguridad",
              "Salir"
            };

            do
            {
                opcion = Util.Menu(opciones, opciones.Length);
                switch (opcion)
                {
                    case 1:
                        emp.SetEmpleado();
                        hospital.SetPersona(emp);
                        break;
                    case 2:
                        enf.SetEnfermo();
                        hospital.SetPersona(enf);
                        break;
                    case 3:
                        System.Console.WriteLine("Introduzca el nombre
de la persona: ");
                        nom = System.Console.ReadLine();
                        hospital.BuscarPersona(nom);
                        break;
                    case 4:
                        hospital.MostrarGente();
                        break;
                    case 5:
                        hospital_copia = new CHospital(hospital);
                        System.Console.WriteLine("Copia realizada...");
                        break;
                    case 6:
                        hospital = hospital_copia;
                        System.Console.WriteLine("Copia
restaurada...");
                        break;
                    case 7:
                        break;
                }
            } while (opcion != 7);
        }
    }
}
```

Solución de la práctica 5

```
/*CTemperatura.cs*/
using System;
namespace Temperatura
{
    public class CTemperatura
    {
        private int valor;
        private CHora HoraRegistro;

        public CTemperatura()
        {
            valor = 0;
            HoraRegistro = new CHora();
        }

        public CTemperatura(int val, int h, int m, int s, string f)
        {
            valor = val;
            HoraRegistro = new CHora(h, m, s, f);
        }
        public virtual CTemperatura clonar()
        {
            return new CTemperatura(this);
        }

        public CTemperatura(CTemperatura t)
        {
            valor = t.valor;
            HoraRegistro = new CHora(t.HoraRegistro);
        }

        public int GetValor()
        {
            return valor;
        }

        public void SetTemperatura()
        {
            int h, m, s;
            string f;
            bool aux = false;

            System.Console.WriteLine("Leer el valor de la
Temperatura");
            valor = Leer.datoInt();
            if (valor < 0 || valor > 100)
                System.Console.WriteLine(ExcTemperatura.Mensaje());

            System.Console.WriteLine("Introduzca la hora de registro de
la temperatura: ");
            do
            {
                System.Console.WriteLine("Introduzca la hora: ");
```

```
                h = Leer.datoInt();
                System.Console.WriteLine("Introduzca los minutos: ");
                m = Leer.datoInt();
                System.Console.WriteLine("Introduzca los segundos: ");
                s = Leer.datoInt();
                do
                {
                    System.Console.WriteLine("Introduzca el formato =>
AM, PM o 24 ");
                    f = System.Console.ReadLine();
                    if (f == "AM" || f == "PM" || f == "24")
                        aux = true;  //Correcto
                    else
                        aux = false;
                    if (aux == false)
                        System.Console.WriteLine("El formato
introducido es incorrecto. ");
                } while (aux == false);

                aux = HoraRegistro.SetHora(h, m, s, f);

            } while (aux == false);
        }

        public virtual void Mostrar()
        {
            System.Console.WriteLine("El valor de la Temperatura es: "
+ valor);
            HoraRegistro.GetHora();
        }
    }
}
```

```
/*ExcTemperatura.cs*/
using System;
namespace Temperatura
{
    public class ExcTemperatura
    {
        public static string Mensaje()
        {
            try
            {
                throw new ArgumentOutOfRangeException("Error : ");
            }
            catch (ArgumentOutOfRangeException e)
            {
                throw new ArgumentOutOfRangeException("Error Valor no
valido." + e.Message, e);
            }
        }
    }
}
```

```
/*CHora.cs*/
using System;
```

```
namespace Temperatura
{
    public class CHora
    {
        private int Horas;
        private int Minutos;
        private int Segundos;
        private string formato;

        public CHora()
        {
            Horas = 0;
            Minutos = 0;
            Segundos = 0;
            formato = "";
        }

        public CHora(int h, int m, int s, string f)
        {
            Horas = h;
            Minutos = m;
            Segundos = s;
            formato = f;
        }

        public CHora(CHora hora)
        {
            Horas = hora.Horas;
            Minutos = hora.Minutos;
            Segundos = hora.Segundos;
            formato = hora.formato;
        }

        public bool Formato()
        {
            if (formato == "AM" || formato == "PM")
                return true;
            else
                return false;
        }

        public bool VerificarHora()
        {
            bool f = false, h = false, m = false, s = false;
            System.Console.WriteLine("Verificando la hora...");
            f = Formato();
            switch (f)
            {
                case false:
                    h = ((Horas >= 0) && (Horas <= 23));
                    m = ((Minutos >= 0) && (Minutos <= 59));
                    s = ((Segundos >= 0) && (Segundos <= 59));
                    break;
                case true:
                    h = ((Horas >= 0) && (Horas <= 11));
                    m = ((Minutos >= 0) && (Minutos <= 59));
                    s = ((Segundos >= 0) && (Segundos <= 59));
```

```
                    break;
            }//fin switch

            if (h && m && s)
            {
                System.Console.WriteLine("La Hora esta CORRECTA...");
                return (true);
            }
            else
            {
                System.Console.WriteLine("La Hora esta INCORRECTA...");
                return (false);
            }
        }

        public bool SetHora(int nHoras, int nMinutos, int nSegundos,
string format)
        {
            Horas = nHoras;
            Minutos = nMinutos;
            Segundos = nSegundos;
            formato = format;
            return VerificarHora();
        }

        public void GetHora()
        {
            System.Console.WriteLine("La hora de registro de la
Temperatura es :");
            System.Console.WriteLine(Horas + ":" + Minutos + ":" +
Segundos + " " + formato);
        }
    }
}
```

```
/*Leer.cs*/
using System;
namespace Temperatura
{
    public class Leer
    {
        public static short datoShort()
        {
            try
            {
                return Int16.Parse(Console.ReadLine());
            }
            catch (FormatException)
            {
                return Int16.MinValue; //valor mas pequeno
            }
        }

        public static int datoInt()
        {
            try
```

```
            {
                return Int32.Parse(Console.ReadLine());
            }
            catch (FormatException)
            {
                return Int32.MinValue; //valor mas pequeno
            }
        }

        public static long datoLong()
        {
            try
            {
                return Int64.Parse(Console.ReadLine());
            }
            catch (FormatException)
            {
                return Int64.MinValue; //valor mas pequeno
            }
        }

        public static float datoFloat()
        {
            try
            {
                return Single.Parse(Console.ReadLine());
            }
            catch (FormatException)
            {
                return Single.NaN; // no es un numero; valor float
            }
        }

        public static double datoDouble()
        {
            try
            {
                return Double.Parse(Console.ReadLine());
            }
            catch (FormatException)
            {
                return Double.NaN; //no es un numero; valor double
            }
        }

        public static int Menu(string[] opciones, int numOpciones)
        {
            int i;
            int opcion = 0;
            Console.WriteLine(" \n
_____________________________________________________\n\n");
            for (i = 1; i <= numOpciones; ++i)
                Console.WriteLine("       " + i + ".-" + opciones[i -
1]);
            Console.WriteLine("
_____________________________________________________\n");
            do
```

```
            {
                Console.WriteLine("\nOpcion (1 - " + numOpciones + "):
");
                opcion = datoInt();
            }
            while (opcion < 1 || opcion > numOpciones);
            return opcion;
        }
    }
}
```

```
/* Test.cs*/
using System;
namespace Temperatura
{
    public class Test
    {

        public static void Main(string[] args)
        {
            CTemperatura temp = new CTemperatura();

            int opcion;

            string[] opciones =
            {
              "Leer temperatura y hora de Registro",
              "Mostrar temperatura y hora de registro",
              "Salir"
            };

            do
            {
                opcion = Leer.Menu(opciones, opciones.Length);
                switch (opcion)
                {
                    case 1:
                        {
                            try
                            {
                                temp.SetTemperatura();
                            }
                            catch (ArgumentOutOfRangeException e)
                            {
                                Console.WriteLine("\n" + e.Message);
                                Console.WriteLine("----------------------
--------------------------");
                            }
                            finally
                            {
                                Console.WriteLine("\n Gracias por usar
nuestro programa");
                            }
                        }
                        break;
                    case 2:
```

```
                        temp.Mostrar();
                        break;
                    case 3:
                        break;
                }
            } while (opcion != 3);
        }
    }
}
```