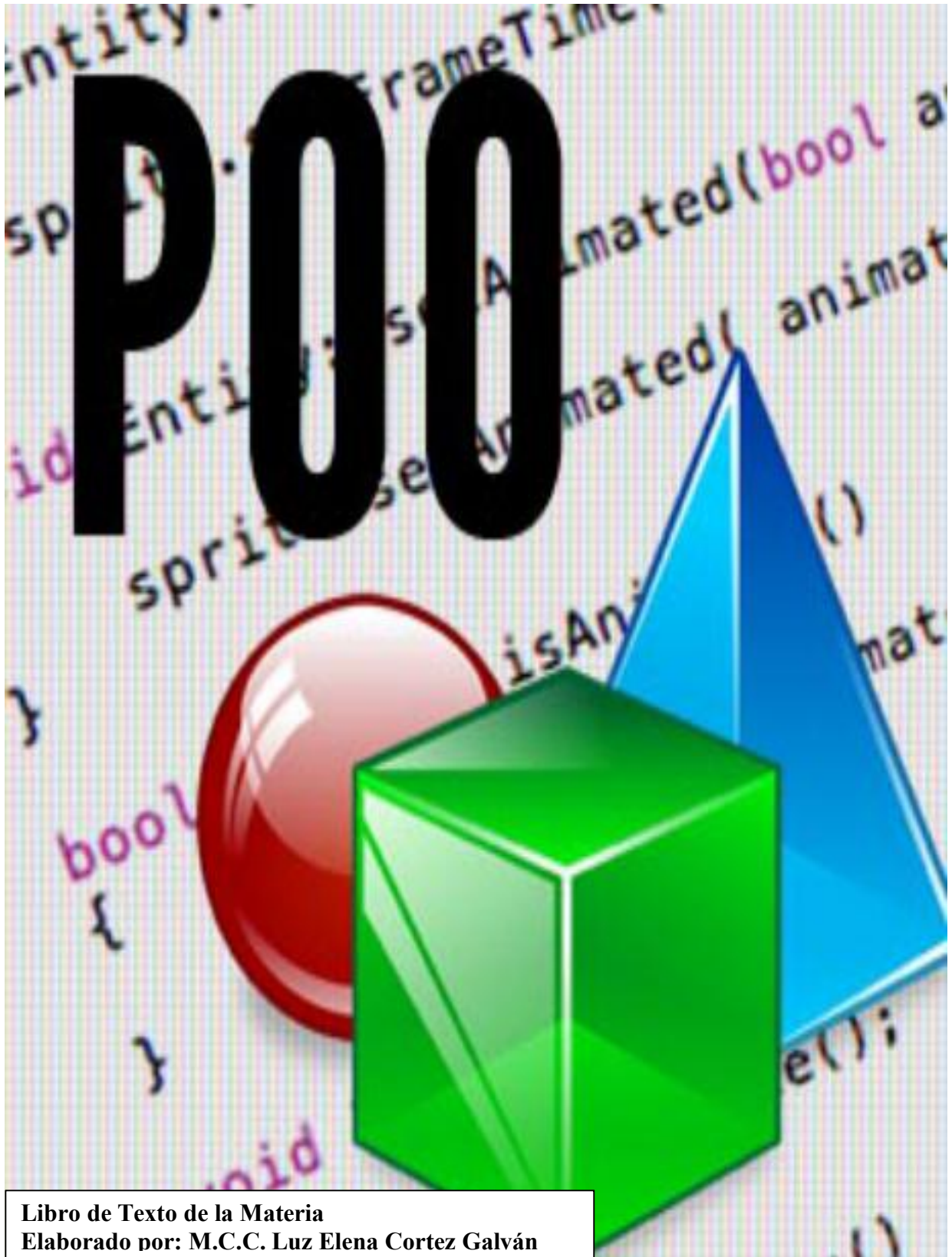




Libro de Texto de la Materia
Elaborado por: M.C.C. Luz Elena Cortez Galván

UNIDAD I – Introducción al Paradigma de la Programación Orientada a Objetos.

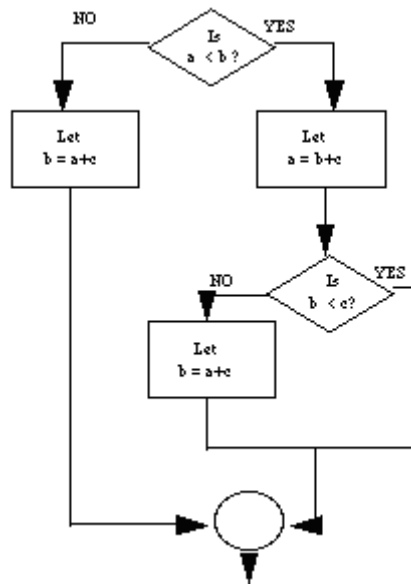
1.1 – Elementos del Modelo de Objetos.

Hoy en día la tecnología orientada a objetos ya no se aplica solamente a los lenguajes de programación, además se viene aplicando en el análisis y diseño con mucho éxito, al igual que en las bases de datos. Es que para hacer una buena programación orientada a objetos hay que desarrollar todo el sistema aplicando esta tecnología, de ahí la importancia del análisis y el diseño orientado a objetos.

La programación orientada a objetos es una de las formas más populares de programar y viene teniendo gran acogida en el desarrollo de proyectos de software desde los últimos años. Esta acogida se debe a sus grandes capacidades y ventajas frente a las antiguas formas de programar.

Una Perspectiva Histórica

Tradicionalmente, la programación fue hecha en una manera secuencial o lineal, es decir una serie de pasos consecutivos con estructuras consecutivas y bifurcaciones.



Los lenguajes basados en esta forma de programación ofrecían ventajas al principio, pero el problema ocurre cuando los sistemas se vuelven complejos. Estos programas escritos al estilo “espagueti” no ofrecen flexibilidad y el mantener una gran cantidad de líneas de código en sólo bloque se vuelve una tarea complicada.

Frente a esta dificultad aparecieron los lenguajes basados en la programación estructurada. La idea principal de esta forma de programación es separar las partes complejas del programa en módulos o segmentos que sean ejecutados conforme se requieran. De esta manera tenemos un diseño modular, compuesto por módulos independientes que puedan comunicarse entre sí. Poco a poco este estilo de programación fue reemplazando al estilo “espagueti” impuesto por la programación lineal.

Entonces, vemos que la evolución que se fue dando en la programación se orientaba siempre a ir descomponiendo más el programa. Este tipo de descomposición conduce directamente a la programación orientada a objetos.

Pues la creciente tendencia de crear programas cada vez más grandes y complejos llevó a los desarrolladores a crear una nueva forma de programar que les permita crear sistemas de niveles empresariales y con reglas de negocios muy complejas. Para estas necesidades ya no bastaba la programación estructurada ni mucho menos la programación lineal. Es así como aparece la programación orientada a objetos (POO).

La POO viene de la evolución de la programación estructurada; básicamente la POO simplifica la programación con la nueva filosofía y nuevos conceptos que tiene. La POO se basa en dividir el programa en pequeñas unidades lógicas de código. A estas pequeñas unidades lógicas de código se les llama objetos. Los objetos son unidades independientes que se comunican entre ellos mediante mensajes.

¿Cuáles son las ventajas de un lenguaje orientado 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.

El Modelo Orientado a Objetos

Para entender este modelo vamos a revisar los siguientes conceptos básicos:

- Objetos
- Clases
- Herencia
- Polimorfismo
- Abstracción
- Encapsulamiento
- Modularidad

1. Objetos

Entender que es un objeto es la clave para entender cualquier lenguaje orientado a objetos.

Existen muchas definiciones que se le ha dado al Objeto. Primero empecemos entendiendo que es un objeto del mundo real. Un objeto del mundo real es cualquier cosa que vemos a nuestro alrededor. Digamos que para leer este artículo lo hacemos a través del monitor y una computadora, ambos son objetos, al igual que nuestro teléfono celular, un árbol o un automóvil.

Analicemos un poco más a un objeto del mundo real, como la computadora. No necesitamos ser expertos en hardware para saber que una computadora está compuesta internamente por varios componentes: la tarjeta madre, el chip del procesador, un disco duro, una tarjeta de video, y otras partes más. El trabajo en conjunto de todos estos componentes hace operar a una computadora.

Internamente, cada uno de estos componentes puede ser sumamente complicado y puede ser fabricado por diversas compañías con diversos métodos de diseño. Pero nosotros no necesitamos saber cómo trabajan cada uno de estos componentes, como saber qué hace cada uno de los chips de la tarjeta madre, o cómo funciona internamente el procesador. Cada componente es una unidad autónoma, y todo lo que necesitamos saber de adentro es cómo interactúan entre sí los componentes, saber por ejemplo si el procesador y las memorias son compatibles con la tarjeta madre, o conocer donde se coloca la tarjeta de video. Cuando conocemos como interaccionan los componentes entre sí, podremos armar fácilmente una computadora.

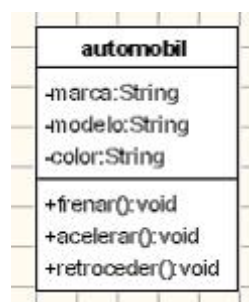
¿Qué tiene que ver esto con la programación? La programación orientada a objetos trabaja de esta manera. Todo el programa está construido en base a diferentes componentes (Objetos), cada uno tiene un rol específico en el programa y todos los componentes pueden comunicarse entre ellos de formas predefinidas.

Todo objeto del mundo real tiene 2 componentes: características y comportamiento.

Por ejemplo, los automóviles tienen **características** (marca, modelo, color, velocidad máxima, etc.) y **comportamiento** (frenar, acelerar, retroceder, llenar combustible, cambiar llantas, etc.).

Los Objetos de Software, al igual que los objetos del mundo real, también tienen características y comportamientos. Un **objeto de software** mantiene sus características en una o más "variables", e implementa su comportamiento con "métodos".

Un método es una función o subrutina asociada a un objeto.



Para complementar estas ideas, imaginemos que tenemos estacionado en nuestra cochera un Ford Focus color azul que corre hasta 260 km/h. Si pasamos ese objeto del mundo real al mundo del software, tendremos un objeto Automóvil con sus características predeterminadas:

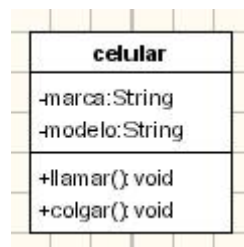
Marca =	Ford
Modelo =	Focus
Color =	Azul
Velocidad Máxima =	260 km/h

Cuando a las características del objeto le ponemos valores, decimos que el objeto tiene estados. Las variables almacenan los estados de un objeto en un determinado momento.

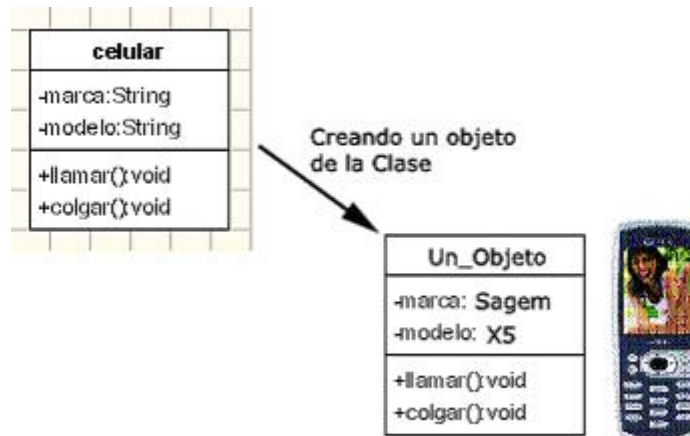
Definición teórica: Un objeto es una unidad de código compuesto de variables y métodos relacionados.

2. Las Clases

En el mundo real, normalmente tenemos muchos objetos del mismo tipo. Por ejemplo, nuestro teléfono celular es sólo uno de los miles que hay en el mundo. Si hablamos en términos de la programación orientada a objetos, podemos decir que nuestro objeto celular es una instancia de una clase conocida como "celular". Los celulares tienen características (marca, modelo, sistema operativo, pantalla, teclado, etc.) y comportamientos (hacer y recibir llamadas, enviar mensajes multimedia, transmisión de datos, etc.).



Cuando se fabrican los celulares, los fabricantes aprovechan el hecho de que los celulares comparten esas características comunes y construyen modelos o plantillas comunes, para que a partir de esas se puedan crear muchos equipos celulares del mismo modelo. A ese modelo o plantilla le llamamos CLASE, y a los equipos que sacamos a partir de ella la llamamos OBJETOS.



Esto mismo se aplica a los objetos de software, se puede tener muchos objetos del mismo tipo y mismas características.

Definición teórica: La clase es un modelo o prototipo que define las variables y métodos comunes a todos los objetos de cierta clase. También se puede decir que una clase es una plantilla genérica para un conjunto de objetos de similares características.

Por otro lado, una instancia de una clase es otra forma de llamar a un objeto. En realidad no existe diferencia entre un objeto y una instancia. Sólo que el objeto es un término más general, pero los objetos y las instancias son ambas representación de una clase.

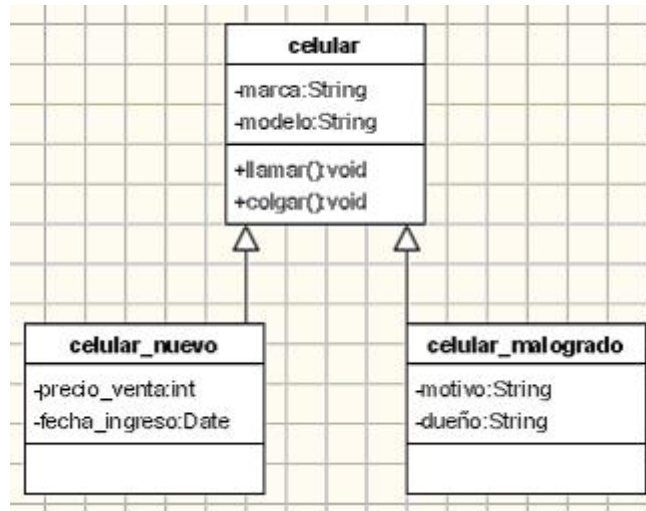
Definición teórica: Una instancia es un objeto de una clase en particular.

3. Herencia

La herencia es uno de los conceptos más cruciales en la POO. La herencia básicamente consiste en que una clase puede heredar sus variables y métodos a varias subclases (la clase que hereda es llamada superclase o clase padre y a la que le hereda, subclase o clase hija). Esto significa que una subclase, aparte de los atributos y métodos propios, tiene incorporados los atributos y métodos heredados de la superclase. De esta manera se crea una jerarquía de clases.

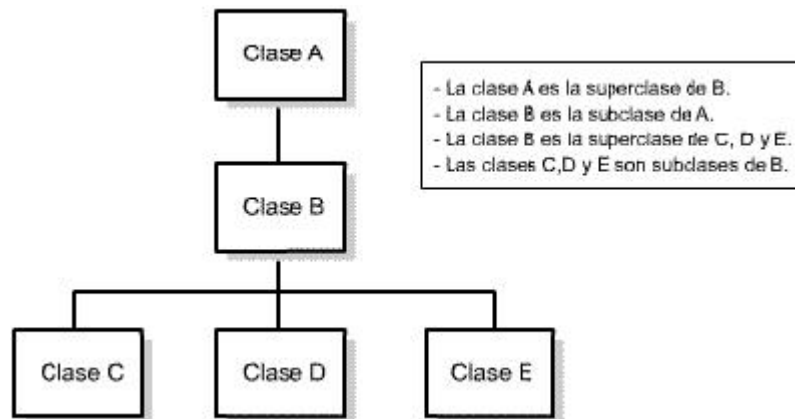
Definición teórica: La herencia es específica de la programación orientada a objetos, donde una clase nueva se crea a partir de una clase ya existente.

Por ejemplo, imaginemos que estamos haciendo el análisis de un Sistema para una tienda que vende y repara equipos celulares.



En la imagen vemos 2 Clases más que posiblemente necesitemos para crear nuestro Sistema. Esas 2 Clases nuevas se construirán a partir de la Clase Celular existente. De esa forma utilizamos el comportamiento de la Superclase.

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



Características asociadas al POO

4. Polimorfismo

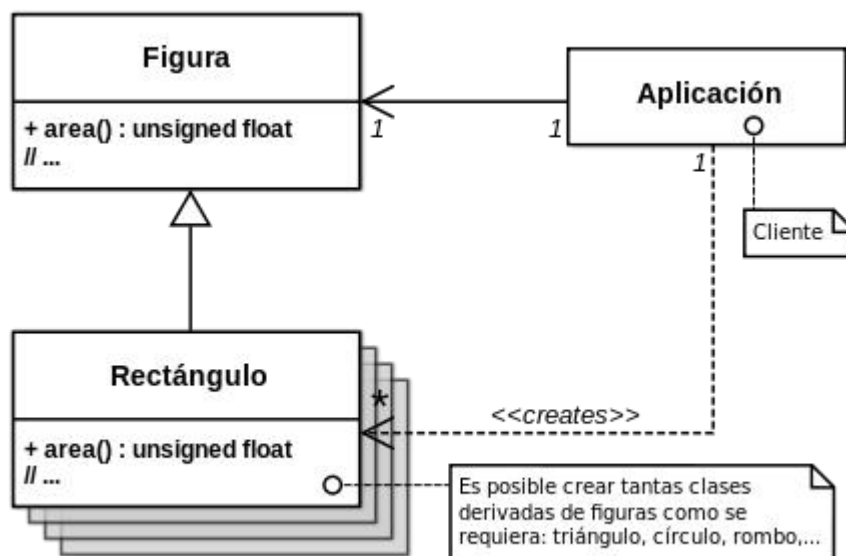
En programación orientada a objetos, el polimorfismo se refiere a la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos. El único requisito que deben cumplir los objetos que se utilizan de manera polimórfica es saber responder al mensaje que se les envía.

En lenguajes basados en clases y con un sistema de tipos de datos fuerte (independientemente de si la verificación se realiza en tiempo de compilación o de

ejecución), es posible que el único modo de poder utilizar objetos de manera polimórfica sea que compartan una raíz común, es decir, una jerarquía de clases, ya que esto proporciona la compatibilidad de tipos de datos necesaria para que sea posible utilizar una misma variable de referencia (que podrá apuntar a objetos de diversas subclases de dicha jerarquía) para enviar el mismo mensaje (o un grupo de mensajes) al grupo de objetos que se tratan de manera polimórfica.

No obstante, algunos lenguajes de programación (Java, C++) permiten que dos objetos de distintas jerarquías de clases respondan a los mismos mensajes, a través de las denominadas interfaces (esta técnica se conoce como composición de objetos). Dos objetos que implementen la misma interfaz podrán ser tratados de forma idéntica, como un mismo tipo de objeto, el tipo definido por la interfaz. Así, distintos objetos podrán intercambiarse en tiempo de ejecución –siempre que sean del mismo tipo–, y además con dependencias mínimas entre ellos. Por estos motivos se considera un buen principio de diseño en programación orientada a objetos el favorecer la composición de objetos frente a la herencia de clases.

Podemos ver un ejemplo práctico en la siguiente imagen:



Otro ejemplo sería crear dos clases distintas: **Pez** y **Ave** que heredan de la superclase **Animal**. La clase **Animal** tiene el método abstracto **mover** que se implementa de forma distinta en cada una de las subclases (peces y aves se mueven de forma distinta). Entonces, un tercer objeto puede enviar el mensaje **mover** a un grupo de objetos **Pez** y **Ave** por medio de una variable de referencia de clase **Animal**, haciendo así un uso polimórfico de dichos objetos respecto del mensaje **mover**.

5. Abstracción

La abstracción consiste en aislar un elemento de su contexto o del resto de los elementos que lo acompañan. En programación, el término se refiere al énfasis en el "¿qué hace?" más que en el "¿cómo lo hace?" (Característica de caja negra). El común denominador en la evolución de los lenguajes de programación, desde los clásicos o imperativos hasta los orientados a objetos, ha sido el nivel de abstracción del que cada uno de ellos hace uso.

La abstracción consiste en captar las características esenciales de un objeto, así como su comportamiento. Por ejemplo, volvamos al ejemplo de los automóviles, ¿Qué características podemos abstraer de los automóviles? O lo que es lo mismo ¿Qué características semejantes tienen todos los automóviles? Todos tendrán una marca, un modelo, número de chasis, peso, llantas, puertas, ventanas, etc. Y en cuanto a su comportamiento todos los automóviles podrán acelerar, frenar, retroceder, etc.

En un esquema POO el automóvil sería el objeto, las propiedades serían las características como el color o el modelo y los métodos serían las funcionalidades asociadas como ponerse en marcha o frenar.

Por poner otro ejemplo vamos a ver cómo modelaríamos en un esquema POO una fracción, es decir, esa estructura matemática que tiene un numerador y un denominador que divide al numerador, por ejemplo $3/2$. La fracción será el objeto y tendrá dos propiedades, el numerador y el denominador. Luego podría tener varios métodos como simplificarse, sumarse con otra fracción o número, restarse con otra fracción, etc.

En los lenguajes de programación orientada a objetos, el concepto de Clase es la representación y el mecanismo por el cual se gestionan las abstracciones.

Por ejemplo, en C# tenemos:

```
public class Automóvil
{
    // variables

    // métodos

    // propiedades
}
```

6. Encapsulamiento

En programación orientada a objetos, se denomina Encapsulamiento al ocultamiento del estado, es decir, de los datos miembro de un objeto de manera que sólo se pueda cambiar mediante las operaciones definidas para ese objeto.

El aislamiento protege a los datos asociados de un objeto contra su modificación por quien no tenga derecho a acceder a ellos, eliminando efectos secundarios e interacciones.

De esta forma el usuario de la clase puede obviar la implementación de los métodos y propiedades para concentrarse sólo en cómo usarlos.

La utilidad del encapsulamiento es la facilidad para manejar la complejidad, ya que tendremos a las Clases como cajas negras donde sólo se conoce el comportamiento pero no los detalles internos, y esto es conveniente porque lo que nos interesará será conocer qué hace la Clase pero no será necesario saber cómo lo hace.

Ocultamiento

Es la capacidad de ocultar los detalles internos del comportamiento de una Clase y exponer sólo los detalles que sean necesarios para el resto del sistema.

El ocultamiento permite 2 cosas: restringir y controlar el uso de la Clase.

Restringir porque habrá cierto comportamiento privado de la Clase que no podrá ser accedido por otras Clases.

Y controlar porque daremos ciertos mecanismos para modificar el estado de nuestra Clase y es en estos mecanismos dónde se validarán que algunas condiciones se cumplan.

En C# el ocultamiento se logra usando las palabras reservadas: **public**, **private** y **protected** antecediendo al nombre de las variables y métodos.

Vamos a ver el ejemplo del vehículo, usaremos la característica COLOR.

Contexto 1: Se necesita que cualquiera pueda acceder al color de un vehículo, entonces:

Opción a: Declaro entonces COLOR como Privado

Opción b: Declaro entonces COLOR como Protegido

Opción c: Declaro entonces COLOR Como Público

Correcta Opción c.

Contexto 2: Se necesita que color se pueda acceder a través no sólo de vehículo, si no ahora también de Camión, y como todos sabemos un camión es un tipo de vehículo, entonces también deberá tener acceso a color.

Opción a: Declaro entonces COLOR como Privado

Opción b: Declaro entonces COLOR como Protegido

Opción c: Declaro entonces COLOR Como Público

Correcta Opción b.

Contexto 3: Se necesita que color se pueda acceder solamente para vehículo.

Opción a: Declaro entonces COLOR como Privado

Opción b: Declaro entonces COLOR como Protegido

Opción c: Declaro entonces COLOR Como Público

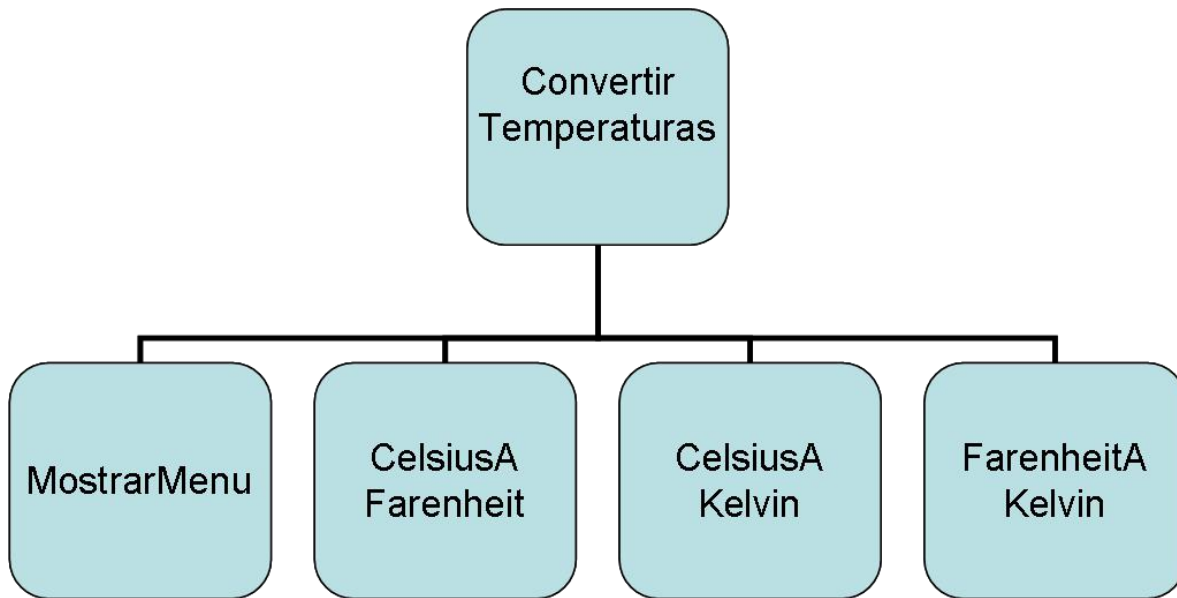
Correcta Opción a.

7. Modularidad

La modularidad es, en programación orientada a objetos, la propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas módulos), cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las partes restantes.

La modularización consiste en dividir un programa en módulos que se puedan compilar por separado, pero que tienen conexiones con otros módulos.

Como se muestra en el siguiente ejemplo el programa principal se divide en varios módulos para simplificar y optimizar su funcionamiento:



La modularidad también tiene principios y son los siguientes:

a) Capacidad de descomponer un sistema complejo.

De acuerdo al principio de "Divide y Vencerás", en este procedimiento se realiza algo similar, ya que se descompone un sistema en subprogramas (llamarlos módulos), el problema en general lo divides en problemas más pequeños.

b) Capacidad de componer a través de sus módulos.

Indica la posibilidad de componer el programa desde los problemas más pequeños completando y resolviendo el problema en general, particularmente cuando se crea software se utilizan algunos módulos existentes para poder formar lo que nos solicitan, estos módulos que se integran a la aplicación deben de ser diseñados para ser reusables.

c) Comprensión de sistema en partes

El poder tener cada parte separada nos ayuda a la comprensión del código y del sistema, también a la modificación del mismo, recordemos que si el sistema necesita modificaciones y no hemos trabajado con módulos definitivamente eso será un caos.

1.2 – Lenguaje de Modelado Unificado.

A mediados de los noventa existían muchos métodos de análisis y diseño Orientado a Objetos.

- ✓ Mismos conceptos con distinta notación
- ✓ Mucha confusión.

En 1994, Booch, Rumbaugh y Jacobson deciden unificar las notaciones de sus métodos creando:

Unified Modeling Language (UML)

El proceso de estandarización fue promovido por el **OMG**:

<http://www.omg.org>

El Object Management Group (OMG) es un consorcio, formado en 1989, dedicado al cuidado y el establecimiento de diversos estándares de tecnologías orientadas a objetos, tales como UML, XMI, CORBA y BPMN.

Es una organización sin fines de lucro que promueve el uso de tecnología orientada a objetos mediante guías y especificaciones.

EVOLUCION DE UML:

- Grady Booch y Jim Rumbaugh comenzaron a unificar sus métodos (Octubre, 1994).
- Borrador de UML (versión 0.8) (Octubre, 1995)
- Ivar Jacobson se une al proyecto (Noviembre, 1995).
- UML 0.9 y se crea un consorcio (Junio, 1996)
- OMG lanza una petición para un lenguaje unificado (1996)
- UML 1.0 es ofrecido al OMG (Enero, 1997)
- Se extiende el consorcio (Enero-Julio, 1997)
- UML 1.1 es ofrecido al OMG (Julio, 1997)
- OMG adopta UML 1.1 (Noviembre, 1997)
- Se crea el UML RTF (1998)
- UML 1.3 (Mayo 1999)
- UML 2.0 (principios de 2005)

UML es un lenguaje para visualizar, especificar, construir y documentar los artefactos (modelos) de un sistema de software, desde una perspectiva orientada a objetos.

MODELOS EN UML:

- Modelado de Casos de Uso
- Modelado Estructural
- Modelado de Comportamiento
- Modelado de flujos de Actividades
- Modelado Implementación
- Modelado de Despliegue

¿Qué es un modelo?

“Un modelo es una simplificación de la realidad”.

“Un modelo es resultado de un proceso de abstracción, y ayuda a comprender y razonar sobre una realidad.”

¿Qué es un modelo de software?

“Es una descripción de un aspecto del sistema, escrita en un lenguaje bien definido”.

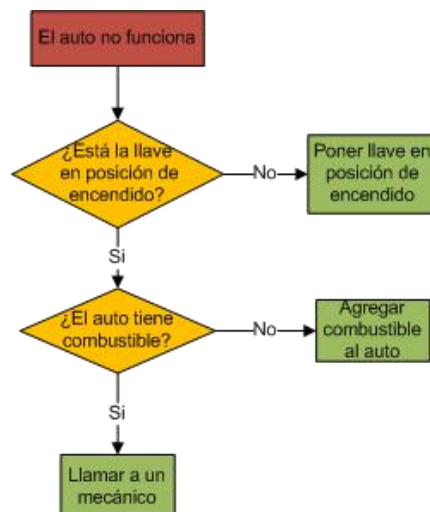
DIAGRAMAS DE UML 2.0

- ❖ Diagrama de Clases
- ❖ Diagrama de Objetos
- ❖ Diagrama de Componentes
- ❖ Diagrama de Estructura Compuesta
- ❖ Diagrama de Casos de Uso
- ❖ Diagrama Secuencia
- ❖ Diagrama Comunicación (antes de Colaboración)
- ❖ Diagrama de Estados
- ❖ Diagrama de Actividades
- ❖ Diagrama de Despliegue
- ❖ Diagrama de Artefactos
- ❖ Diagrama de Paquetes
- ❖ Diagrama de Tiempos

Diagramas no
son modelos

¿Qué es un diagrama?

Diagrama (del latín, diagramma; y este del griego διάγραμμα, «diseño») es un dibujo geométrico, muy utilizado en ciencia, en educación y en comunicación; con el que se obtiene la presentación gráfica de una proposición, de la resolución de un problema, de las relaciones entre las diferentes partes o elementos de un conjunto o sistema, o de la regularidad en la variación de un fenómeno que permite establecer algún tipo de ley.



¿Qué es un diagrama de clases?

En ingeniería de software, un diagrama de clases en Lenguaje Unificado de Modelado (UML) es un tipo de diagrama de estructura estática que describe la estructura de un sistema mostrando las clases del sistema, sus atributos (características), operaciones (métodos), y las relaciones entre los objetos.

Un diagrama de clases esta compuesto por los siguientes elementos:

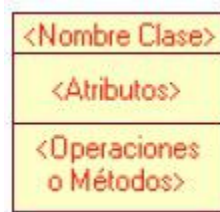
- **Clase:** atributos, métodos y visibilidad.
- **Relaciones:** Herencia, Composición, Agregación, Asociación y Uso.

Elementos

- **Clase**

Es la unidad básica que encapsula toda la información de un Objeto (un objeto es una instancia de una clase). A través de ella podemos modelar el entorno en estudio (una Casa, un Auto, una Cuenta Corriente, etc.).

En UML, una clase es representada por un rectángulo que posee tres divisiones:



En donde:

El Nivel Superior: Contiene el nombre de la Clase

El Nivel Intermedio: Contiene los atributos (o variables de instancia) que caracterizan a la Clase (pueden ser private, protected o public).

El Nivel Inferior: Contiene los métodos u operaciones, los cuales son la forma como interactúan el objeto con su entorno (dependiendo de la visibilidad: private, protected o public).

Ejemplo practico:

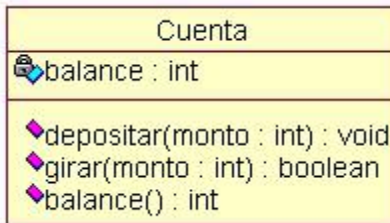
Una Cuenta Corriente que posee como característica:

- ✓ Balance

Puede realizar las operaciones de:

- ✓ Depositar
- ✓ Girar
- ✓ y Balance

El diseño asociado es:



Atributos:

Los atributos o características de una Clase pueden ser de tres tipos, los que definen el grado de comunicación y visibilidad de ellos con el entorno, estos son:

- public (+,): Indica que el atributo será visible tanto dentro como fuera de la clase, es decir, es accesible desde todos lados.
- private (-,): Indica que el atributo sólo será accesible desde dentro de la clase (sólo sus métodos lo pueden acceder).
- protected (#,): Indica que el atributo no será accesible desde fuera de la clase, pero si podrá ser accesado por métodos de la clase además de las subclases que se deriven de ella.

Métodos:

Los métodos u operaciones de una clase son la forma en como ésta interactúa con su entorno, éstos pueden tener las características:

- public (+,): Indica que el método será visible tanto dentro como fuera de la clase, es decir, es accesible desde todos lados.
- private (-,): Indica que el método sólo será accesible desde dentro de la clase (sólo otros métodos de la clase lo pueden acceder).
- protected (#,): Indica que el método no será accesible desde fuera de la clase, pero si podrá ser accesado por métodos de la clase además de métodos de las subclases que se deriven de ella.

● Relaciones entre Clases:

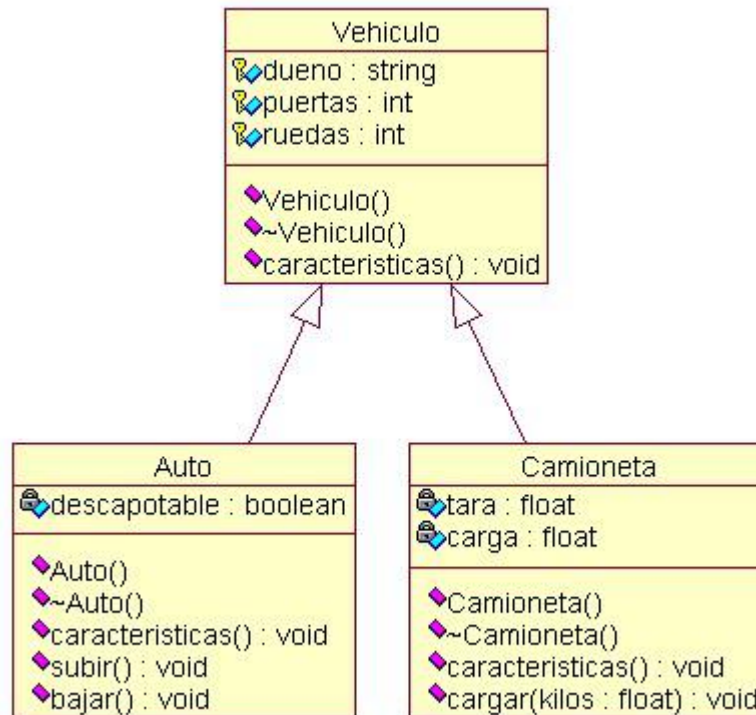
Ahora ya se definió el concepto de Clase, es necesario explicar como se pueden interrelacionar dos o más clases (cada una con características y objetivos diferentes).

Antes es necesario explicar el concepto de cardinalidad de relaciones: En UML, la cardinalidad de las relaciones indica el grado y nivel de dependencia, se anotan en cada extremo de la relación y éstas pueden ser:

- uno a muchos: 1..* (1..n)
- 0 a muchos: 0..* (0..n)
- número fijo: m (m denota el número).

a) **Herencia (Especialización/Generalización):**

Indica que una subclase hereda los métodos y atributos especificados por una Super Clase, por ende la subclase además de poseer sus propios métodos y atributos, poseerá las características y atributos visibles de la Super Clase (public y protected), ejemplo:



En la figura se especifica que Auto y Camioneta heredan de Vehículo, es decir, Auto posee las Características de Vehículo (Precio, VelMax, etc) además posee algo particular que es Descapotable, en cambio Camioneta también hereda las características de Vehículo (Precio, VelMax, etc) pero posee como particularidad propia Tara y Carga.

Cabe destacar que fuera de este entorno, lo único "visible" es el método Características aplicable a instancias de Vehículo, Auto y Camioneta, pues tiene definición publica, en cambio atributos como Descapotable no son visibles por ser privados.

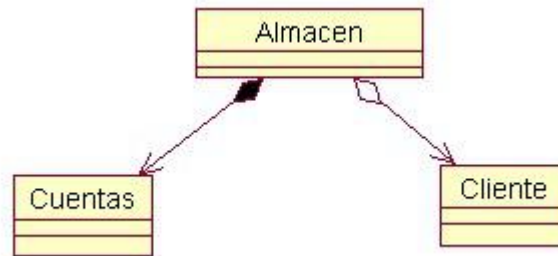
b) **Agregación:**

Para modelar objetos complejos, bastan los tipos de datos básicos que proveen los lenguajes: enteros, reales y secuencias de caracteres. Cuando se requiere componer objetos que son instancias de clases definidas por el desarrollador de la aplicación, tenemos dos posibilidades:

- **Por Valor:** Es un tipo de relación estática, en donde el tiempo de vida del objeto incluido esta condicionado por el tiempo de vida del que lo incluye. Este tipo de relación es comúnmente llamada Composición (el Objeto base se construye a partir del objeto incluido, es decir, es "parte/todo").
- **Por Referencia:** Es un tipo de relación dinámica, en donde el tiempo de vida del objeto incluido es independiente del que lo incluye. Este tipo de relación es

comúnmente llamada Agregación (el objeto base utiliza al incluido para su funcionamiento).

Un Ejemplo es el siguiente:



En donde se destaca que:

- Un Almacén posee Clientes y Cuentas (los rombos van en el objeto que posee las referencias).
- Cuando se destruye el Objeto Almacén también son destruidos los objetos Cuenta asociados, en cambio no son afectados los objetos Cliente asociados.
- La composición (por Valor) se destaca por un rombo relleno.
- La agregación (por Referencia) se destaca por un rombo transparente.

La flecha en este tipo de relación indica la navegabilidad del objeto referenciado. Cuando no existe este tipo de particularidad la flecha se elimina.

c) Asociación:

La relación entre clases conocida como Asociación, permite asociar objetos que colaboran entre si. Cabe destacar que no es una relación fuerte, es decir, el tiempo de vida de un objeto no depende del otro.

Ejemplo:



Un cliente puede tener asociadas muchas Ordenes de Compra, en cambio una orden de compra solo puede tener asociado un cliente.

d) Dependencia o Instanciación (uso):

Representa un tipo de relación muy particular, en la que una clase es instanciada (su instanciación es dependiente de otro objeto/clase). Se denota por una flecha punteada.

El uso más particular de este tipo de relación es para denotar la dependencia que tiene una clase de otra, como por ejemplo una aplicación gráfica que instancia una ventana (la creación del Objeto Ventana esta condicionado a la instanciación proveniente desde el objeto Aplicación):



Cabe destacar que el objeto creado (en este caso la Ventana gráfica) no se almacena dentro del objeto que lo crea (en este caso la Aplicación).

REFERENCIAS BIBLIOGRAFICAS UNIDAD 1

Enciclopedia de Programación Es.ccm.net

Wikipedia, la enciclopedia libre <https://es.wikipedia.org/wiki/Abstracci3n>

Modularidad en la programación orientada a objetos – styde.net
<https://styde.net/modularidad-en-la-programacion-orientada-a-objetos/>

G. Booch, J. Rumbaugh, I. Jacobson, “El lenguaje unificado de modelado”, 2ª Edición, Addison-Wesley, 2006.

Craig Larman, “UML y Patrones: Una introducción al análisis y diseño orientado a objetos y al proceso unificado”, Prentice-Hall, 2003.

Jim Arlow, Ila Neustadt, “UML 2”, Anaya Multimedia, 2006.

Diagrama y Diagrama de clases https://es.wikipedia.org/wiki/Diagrama_de_clases

Tutorial de UML - Modelo de Clases users.dcc.uchile.cl/~psalinas/uml/modelo.html

UNIDAD II – Clases y Objetos.

2.1 – Declaración de clases: atributos, métodos, encapsulamiento.

¿Qué es una clase?

Una clase es algo abstracto que define la "forma" del objeto, se podría hablar de la clase como el molde para crear los objetos.

Podemos describir también una clase como una plantilla que define variables y métodos comunes para todos los objetos de cierto tipo.

■ Declaración de clases

Las clases en C# se declaran mediante la palabra clave **class**, un nombre, y un par de llaves.

Los datos y los métodos de la clase forman el cuerpo de la clase escritos entre las llaves.

El cuerpo de una clase contiene métodos ordinarios y campos (las variables en una clase son llamadas campos). Como se muestra en el ejemplo siguiente:

```
public class Customer
{
    //Fields, properties, methods and events go here...
}
```

El nivel de acceso precede a la palabra clave class. En este caso, se utiliza public, que significa que cualquiera puede crear objetos a partir de esta clase.

El nombre de la clase sigue a la palabra clave class. En este ejemplo, **Customer**.

El resto de la definición es el cuerpo de clase, donde se definen el comportamiento y los datos. Los campos, propiedades, métodos y eventos de una clase se conocen colectivamente como miembros de la clase.

¿Qué es un atributo?

Las propiedades o atributos son las características de los objetos. Cuando definimos una propiedad normalmente especificamos su nombre y su tipo. Nos podemos hacer a la idea de que las propiedades son algo así como variables donde almacenamos datos relacionados con los objetos.

■ Atributos:

Los atributos son las características individuales que diferencian un objeto de otro y determinan su apariencia, estado u otras cualidades.

Los atributos se guardan en variables denominadas de instancia, y cada objeto particular puede tener valores distintos para estas variables.

Las variables de instancia también denominados miembros dato, son declaradas en la clase pero sus valores son fijados y cambiados en el objeto.

Además de las variables de instancia hay variables de clase, las cuales se aplican a la clase y a todas sus instancias. Por ejemplo, el número de ruedas de un automóvil es el mismo cuatro, para todos los automóviles.

■ Los miembros dato:

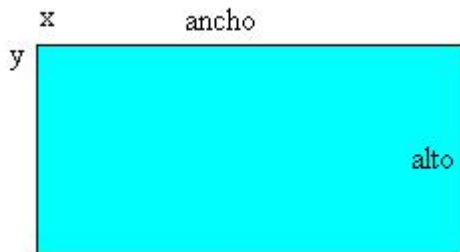
Los valores de los atributos se guardan en los miembros dato o variables de instancia.

Los nombres de dichas variables comienzan por letra minúscula.

Por ejemplo, vamos a crear una clase denominada Rectángulo, que describa las características comunes a estas figuras planas que son las siguientes:

El origen del rectángulo: el origen o posición de la esquina superior izquierda del rectángulo en el plano determinado por dos números enteros x e y.

Las dimensiones del rectángulo: ancho y alto, otros dos números enteros.



```
public class Rectángulo{
    int x;
    int y;
    int ancho;
    int alto;
    //aquí seguirían las funciones (métodos) miembro
}
```

¿Qué es un método?

En programación, un método es una subrutina cuyo código es definido en una clase y puede pertenecer tanto a una clase, como es el caso de los métodos de clase o estáticos, como a un objeto, como es el caso de los métodos de instancia.

■ Comportamiento:

El comportamiento de los objetos de una clase se implementa mediante funciones miembro o métodos.

Un método es un conjunto de instrucciones que realizan una determinada tarea y son similares a las funciones de los lenguajes estructurados.

Del mismo modo que hay variables de instancia y de clase, también hay métodos de instancia y de clase. En el primer caso, un objeto llama a un método para realizar una determinada tarea, en el segundo, el método se llama desde la propia clase.

■ Funciones miembro:

El nombre de las funciones miembro o métodos comienza por una letra minúscula y deben sugerir acciones (mover, calcular, etc.). La definición de una función tiene el siguiente formato:

```
tipo nombreFuncion(tipo parm1, tipo parm2,....., tipo parmN)
{
    //...sentencias
}
```

Entre las llaves de apertura y cierre se coloca la definición de la función, tipo indica el tipo de dato que devolverá la función, el cual puede ser predefinido: int, double, etc, o definido por el usuario, el tipo de una clase cualquiera.

Siguiendo con el ejemplo; hemos definido los atributos o miembros dato de la clase Rectángulo, ahora le vamos añadir un comportamiento: los objetos de la clase Rectángulo, sabrán calcular su área y tendrán la capacidad para trasladarse a otro punto del plano.

a) La función que calcula el área realizará la siguiente tarea, calculará el producto del ancho por el alto del rectángulo y devolverá el resultado. La función devuelve un entero es por tanto, de tipo int. No es necesario pasarle datos ya que tiene acceso a los miembros dato ancho y alto que guardan la anchura y la altura de un rectángulo concreto.

```
public class Rectángulo{
    //datos
    int x;
    int y;
    int ancho;
    int alto;
    //método
    int calcularArea(){
```

```
        return (ancho*alto);  
    }  
}
```

b) A la función que desplaza el rectángulo horizontalmente en dx, y verticalmente en dy, le pasamos dichos desplazamientos, y a partir de estos datos actualizará los valores que guardan sus miembros dato x e y. La función no devuelve nada es de tipo void.

```
public class Rectángulo{  
    //datos  
    int x;  
    int y;  
    int ancho;  
    int alto;  
    //método  
    void desplazar(int dx, int dy){  
        x+=dx;  
        y+=dy;  
    }  
}
```

¿Qué es un el encapsulamiento?

En programación orientada a objetos, se denomina Encapsulamiento al ocultamiento del estado, es decir, de los datos miembro de un objeto de manera que sólo se pueda cambiar mediante las operaciones definidas para ese objeto.

Cada objeto está aislado del exterior, es un módulo natural, y la aplicación entera se reduce a un agregado o rompecabezas de objetos. El aislamiento protege a los datos asociados de un objeto contra su modificación por quien no tenga derecho a acceder a ellos, eliminando efectos secundarios e interacciones.

De esta forma el usuario de la clase puede obviar la implementación de los métodos y propiedades para concentrarse sólo en cómo usarlos. Por otro lado se evita que el usuario pueda cambiar su estado de maneras imprevistas e incontroladas.

■ Niveles de acceso:

La encapsulación define los niveles de acceso para elementos de una clase. Estos niveles de acceso definen los derechos de acceso para los datos, permitiéndonos el

acceso a datos a través de un método de esa clase en particular, desde una clase heredada o incluso desde cualquier otra clase. Existen tres niveles de acceso:

a) público: funciones de toda clase pueden acceder a los datos o métodos de una clase que se define con el nivel de acceso público. Este es el nivel de protección de datos más bajo.

b) protegido: el acceso a los datos está restringido a las funciones de clases heredadas, es decir, las funciones miembro de una clase y todas sus subclases.

c) privado: el acceso a los datos está restringido a los métodos de una clase en particular. Este es nivel más alto de protección de datos.

Ejemplo 1: acceso publico.

Imaginemos que creamos una clase con variables de instancia públicas a las cuales podemos acceder sin problemas desde fuera de la misma clase.

```
public class MiClase {  
    public int tipo;  
}  
  
class AccesoDirecto {  
    public static void main(String[] args) {  
        MiClase mc = new MiClase();  
        mc.tipo = -5; //1  
    }  
}
```

Ejemplo 2: acceso privado.

El ejemplo anterior modificado para un encapsulamiento privado, quedaría como en el siguiente ejemplo; donde para tener acceso al campo de tipo privado, deberemos hacer uso de los métodos set (establecer valor) y get (obtener valor):

```

public class MiClase{
    private int tipo;

    public void setTipo(int t) {
        tipo = t;
    }

    public int getTipo() {
        return tipo;
    }
}

class AccesoIndirecto {

    public static void main(String[] args) {

        MiClase mc = new MiClase();
        mc.setTipo(5);
        System.out.println("El tipo es:" + mc.getTipo());
    }
}

```

Ejemplo 3: acceso protegido.

En el siguiente ejemplo se tienen dos clases; la clase base llamada A y la clase derivada llamada B, la cual puede hacer uso de los campos y métodos que fueron declarados en la clase base.

```

class Program
{
    class A
    {
        protected int X = 12345;

        public void ImprimeA()
        {
            Console.WriteLine("El valor del campo X en la clase A es de: " + X);
        }
    }
}

```

```
class B : A
{
    public void ImprimeB()
    {
        X = 678910;
        Console.WriteLine("El valor del campo X en la clase B es de: " + X);
    }
}

static void Main(string[] args)
{
    A a1 = new A();
    a1.ImprimeA();
    Console.WriteLine("\n\n");
    B b1 = new B();
    b1.ImprimeA();
    b1.ImprimeB();
}
}
```

2.2 – Instanciación de una clase.

Como hemos comentado una clase es el tipo que tendrán nuestros objetos, también llamados instancias de esa clase, es decir, la clase es la forma que tendrán los objetos empleados por nuestra aplicación.

Un ejemplo sencillo puede ser el plano de una casa, el plano sería nuestra clase, pero con ese mismo plano podremos construir diversas casas que serían los objetos. El plano en sí no es nada es solamente la especificación de lo que será una casa una vez construida.

Cuando creamos una instancia de la clase en ejecución tendremos las variables de instancia, una por cada atributo, y todas integradas dentro de la misma instancia. Todas las instancias de la misma clase tendrán los mismos atributos (nombre y tipo), pero cada una de ellas irá tomando distintos valores para ellos. Es decir, tendremos una copia de los atributos por cada objeto de una clase que creemos.

Por ejemplo, tuCoche sería una instancia de la clase Coche y daremos a la variable de instancia velocidadMaxima el valor 180. Mientras que la instancia cocheDeMiguel tiene su propia variable velocidadMaxima con el valor 300.

La funcionalidad de la clase, es la concreción de qué puede hacer cada una de las instancias de la clase, cómo cambian su estado, o cómo se relaciona con otras.

Para nuestro ejemplo, en la clase Coche podríamos determinar la siguiente funcionalidad:

- Arrancar
- Parar (el motor)
- Acelerar
- Desacelerar
- Girar
- Frenar.....

Y podría quedar de la siguiente manera:

```
class Coche {

    String color, modelo;
    int velocidadMaxima;
    boolean motorEncendido;

    Coche(String sColor, String sModelo, int iVelocidadMaxima) {
        color=sColor;
        modelo=sModelo;
        velocidadMaxima=iVelocidadMaxima;
        motorEncendido=false;
    }

    void arrancar() {
        if (motorEncendido == true)
            System.out.println( "El motor ya estaba
encendido" );
        else {
            motorEncendido = true;
            System.out.println( "El motor está ahora
encendido" );
        }
    }

    void visualizar() {
        System.out.println( "Este coche es un " + modelo + ", de
color " + color );
        if (motorEncendido == true)
            System.out.println( "El motor está
encendido." );
        else    System.out.println( "El motor está apagado." );
    }
}
```

Bien, desde algún punto del programa crearemos una instancia de Coche llamada tuCoche. Lo hacemos con el operador *new* con el siguiente formato:

Nombre de la clase	Nombre de la instancia	=	<i>new</i>	Constructor(); de la clase
-----------------------	---------------------------	---	------------	-------------------------------

Coche tuCoche = *new* Coche("verde", "Seat 600", 180);

Una vez ejecutada esta sentencia ya está creada la instancia de la clase.

Para acceder a sus atributos simplemente se indican después del nombre de la instancia y un punto:

```
tuCoche.color = "amarillo con lunares rojos";
```

Con la misma sintaxis se accede a los métodos como si fueran llamadas a funciones:

```
tuCoche.arrancar();
```

```
tuCoche.visualizar();
```

2.3 – Referencia al objeto actual.

this

Al acceder a variables de instancia de una clase, la palabra clave `this` hace referencia a los miembros de la propia clase en el objeto actual; es decir, `this` se refiere al objeto actual sobre el que se está actuando en un método y se utiliza siempre que se quiera hacer referencia al objeto actual de la clase.

En un método, la referencia `this` puede utilizarse en forma implícita y explícita para hacer referencia a las variables de instancia y otros métodos del objeto en el cual se haya llamado.

Por ejemplo, el constructor de la clase `Persona` se puede escribir así usando `this`:

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

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

Utilizando en otro ejemplo una clase llamada `MiClase`, se puede añadir un constructor de la forma siguiente:

```
public class MiClase {
    int i;
    public MiClase() {
        i = 10;
    }
    // Este constructor establece el valor de i

    public MiClase( int valor) {
```

```

        this.i = valor; // i = valor
    }
    // Este constructor también establece el valor de i

    public MiClase( int i ) {
        this.i = i;
    }

}

```

Aquí `this.i` se refiere al entero `i` en la clase `MiClase`, que corresponde al objeto actual. La utilización de `this` en el tercer constructor de la clase, permite referirse directamente al objeto en sí, en lugar de permitir que el ámbito actual defina la resolución de variables, al utilizar `i` como parámetro formal y después `this` para acceder a la variable de instancia del objeto actual.

La utilización de `this` en dicho contexto puede ser confusa en ocasiones, y algunos programadores procuran no utilizar variables locales y nombres de parámetros formales que ocultan variables de instancia. Una filosofía diferente dice que en los métodos de inicialización y constructores, es bueno seguir el criterio de utilizar los mismos nombres por claridad, y utilizar `this` para no ocultar las variables de instancia.

Ejemplo de programa utilizando **This** en java

//Uso implícito y explícito de `this` para hacer referencia a los miembros de un objeto

```

public class PruebaThis
{
    public static void main (String args[])
    {
        TiempoSimple tiempo=new TiempoSimple(15, 30, 19 );
        System.out.println(tiempo.crearString());
    } //fin de main
} //fin de la clase PruebaThis

//la clase TiempoSimple demuestra la referencia "this"
public class TiempoSimple
{
    private int hora; //0-23
    private int minuto; //0-59
}

```

```

private int segundo;//0-59
//si el constructor utiliza nombres de parametros idénticos a
//los nombres de las variables de instancia, se requiere la
//referencia "this" para diferenciar unos nombres de otros
public TiempoSimple (int hora,int minuto, int segundo)
{
this.hora=hora;//establece la hora del objeto "this"
this.minuto=minuto;//establece el minuto del objeto "this"
this.segundo=segundo;//establece el segundo del objeto "this"
} //fin del constructor de TiempoSimple

//usa la referencia "this" explicita e implícita para llamar
StringUniversal
public String crearString()
{
return String.format("%24s:%s\n%24s:%s",
this.aStringUniversal(),this.aStringUniversal(),
"aStringUniversal()",aStringUniversal());
} //fin del metodo crearString
//convierte a String en formato de hora universal (HH:MM:SS)

public String aStringUniversal()
{
//"this"no se requiere aquí para acceder a las variables de
//instancia,ya que el método no tiene variable locales con los
//mismos nombres que las variables de instancia
return String.format("%02d:%02d:%02d",
this.hora, this.minuto, this.segundo);
} //fin del metodo sStringUniversal
} //fin de la clase TiempoSimple

```

2.4 – Métodos: Declaración, mensajes, paso de parámetros, retorno de valores.

Los métodos o funciones miembro se definen dentro de la clase a la que pertenecen y constituyen la interfaz o forma de acceder a la estructura interna de los objetos es decir a los datos privados.

Los métodos definen cual son las operaciones que se pueden realizar con los atributos de los objetos de la clase. La ejecución de un programa orientado a objetos consiste, en recibir, interpretar y responder los objetos a los mensajes que envían otros objetos. En P.O.O. un mensaje está asociado siempre con un método, de manera que cuando un objeto recibe un mensaje la respuesta a ese mensaje es ejecutar el método asociado.

Declaración:

Formato general para la definición de un método

```
[modificadorAcceso] tipoValorRetorno nombreMétodo ([ Tipo1 Parámetro1,
Tipo2 Parámetro2, ...]) //Declaración o cabecera del método
{
    //Cuerpo del método
    Definición de variables locales;    //Sólo accesibles desde el mismo
    método
    Sentencias;        //Instrucciones

    [return ([] expresión []); // Puede no ser la última sentencia, y puede
    aparecer más de una vez en el método
}
```

● Declaración o cabecera del método:

Modificador de acceso: Especifica el tipo de acceso permitido indicando que usuarios de la clase podrán acceder a ese método, los métodos son la única forma de acceso a los atributos privados. Existen varios tipos de acceso.

- public: Accesible desde cualquier otra clase.
- package: Accesible sólo desde el mismo paquete.
- protected: Accesible desde la misma clase y de sus subclases.
- private: Sólo accesible a través de métodos de la propia clase.

Tipo del valor de retorno: Un método puede devolver un valor a quien lo llama o no devolver nada. El valor devuelto por un método puede ser de un tipo primitivo de datos o

una referencia, pero nunca puede devolver más de un valor. El valor de retorno nunca puede ser un objeto de una superclase, sí de la misma clase o de una subclase. Si el método no devuelve nada el tipo devuelto por el método deberá ser el tipo **void**.

Nombre del método: Por convenio, los nombres de los métodos comienzan con minúscula. Si el nombre del método es un nombre compuesto cada nueva palabra empieza con mayúsculas. Los nombres de los métodos suelen ser verbos asociados a la tarea que realizan.

Posible lista de parámetros: la lista de parámetros formales es opcional, la función podría no tenerlos, en caso de que los haya se trata de variables locales (sólo accesibles y visibles desde el propio método) separadas por comas, en las que se debe de especificar el tipo y nombre de cada una, se inicializan en la llamada recibiendo los valores especificados por los argumentos de la llamada. Aunque la función no lleve parámetros hay que poner los paréntesis.

- **Cuerpo del método:**

Definición de variables locales: Dentro de los métodos se pueden definir variables que sólo son accesibles dentro del método en donde se han definido. Este tipo de variables no se inicializan por defecto, pero deberán de inicializarse antes de utilizarlas pues sino el compilador detectara un error.

Sentencias: Instrucciones necesarias para realizar una determinada tarea.

La instrucción return: Devuelve el control de la ejecución al punto del programa donde se hizo la llamada al método. Si el método no devuelve nada y se elimina la sentencia return la función termina con la llave final o llave de cierre de la función, en este caso el tipo devuelto por el método es el tipo void. Si el método retorna un valor, la función no puede ser del tipo void y la sentencia return además de producir la salida de la función, especifica el valor de retorno del método. En una función puede haber más de una sentencia return pero sólo se ejecuta una.

Mensajes:

De forma abstracta, se llama mensaje a una porción de información que un proceso emisor envía a un destinatario (El cual puede ser otro proceso, un actor o un objeto). El

modelo de paso de mensajes es el que definen los métodos y funciones para poder llevar a cabo el envío de un mensaje de un proceso emisor a un destinatario.

El paso de mensajes es una técnica que se usa para, desde un proceso, invocar de forma abstracta un comportamiento concreto por parte de otro actor (Por ejemplo, ejecutar una función o un programa). El emisor envía un mensaje solicitando una acción abstracta al destinatario, y es el destinatario el que, al recibir el mensaje, decidirá la acción que debe ejecutar para cumplir la petición.

Entre las ventajas que proporciona este mecanismo destacan el alto nivel de encapsulamiento y la distribución. En programación, la filosofía de encapsular la información consiste en que los objetos deben poder invocar servicios a otros objetos, sin tener por qué conocer ni preocuparse ni de cómo ni de qué manera está implementada en su interior la resolución de dicha demanda. Esto deriva en un código más limpio, corto y fácil de comprender y mantener.

Caso práctico en programación orientada a objetos:

Un ejemplo clásico de aplicación del modelo de paso de mensajes podría ser en el ámbito informática, si el programador está creando un programa que trabaja con diversos tipos de figuras geométricas de un clase, el programa contendrá varios objetos instancias de dicha clase, como por ejemplo cuadrado, rombo o triángulo.

Imaginemos el caso en que el programador desea conocer el área de los objetos de la clase declarados. Como el cálculo del área de dichas figuras no es el mismo para todas, cada una tendrá un algoritmo matemático distinto para calcularla, en un modelo tradicional el programador se vería obligado a crear una estructura condicional IF-ELSE que consultase una a una el tipo de forma de cada objeto para ejecutar la operación matemática correspondiente.

En lugar de eso, en el modelo de paso de mensajes el programador simplemente debe enviar un mensaje a cada objeto consultando su área, y sería la implementación interna de la clase la que se encargaría de escoger y calcular de forma correcta dicho dato. El objeto respondería a la petición enviando de vuelta otro mensaje al destinatario con el área resultante del cálculo. Como es natural, para que esto sea posible, la clase destinataria debe tener implementado previamente un método por el cual pueda interpretar los mensajes de forma automatizada, así como las funciones necesarias para llevar a cabo el cálculo del área de cada una de las figuras.

Llamada a un método:

Para que un método se ejecute hay que llamarlo. La llamada o invocación a un método provoca la ejecución de las instrucciones que lo componen, una vez se han ejecutado el control de la ejecución vuelve a quien hizo la llamada.

La llamada al método consta de:

- Nombre del método.
- Posible lista de argumentos, llamados también parámetros actuales, entre paréntesis y separados por comas.

Si un método no está definido como static la llamada o invocación al método hay que hacerla a través de un objeto que pertenezca a la clase que define el método y se dice que se envía un mensaje al objeto.

Formato general para llamar un método

```
nombreObjeto.nombreMétodo([lista de argumentos]) //El método se aplica al  
objeto de la clase a través del operador punto (.)
```

En este caso el objeto es siempre el argumento implícito del objeto. Los métodos pueden tener además argumentos explícitos que son los que van entre paréntesis a continuación del nombre del método.

Cuando se accede al método de un objeto se interpreta que el objeto ha recibido un mensaje y el objeto responde al mensaje ejecutando el método.

Los mensajes que puede recibir un objeto se corresponden con el nombre de los métodos de su clase.

Si el método devuelve un valor, la llamada a la función debe de formar parte de una expresión que reciba el valor retornado por la función.

Paso de parámetros a una función o método:

Los parámetros de una función son variables locales que se inicializan en el momento de la llamada al método. Fuera de la función no se conocen y no pueden ser accedidas. Se crean al entrar en la función y se destruyen al salir de ella.

El paso de parámetros o argumentos a las funciones se puede hacer de dos formas:

- Paso por valor.
- Paso por referencia.

Paso por valor: Los parámetros de la función reciben una copia del valor de los argumentos de la llamada.

Las modificaciones que se hagan sobre los parámetros formales no afectan al valor de los parámetros actuales.

Paso por dirección o referencia: Los parámetros formales reciben una copia de la referencia a un objeto. Los parámetros formales serán por tanto de tipo referenciado.

El método actúa directamente sobre el objeto, si modifica sus datos, una vez que termine la ejecución del método, los objetos quedaran modificados.

Los métodos se declaran en una clase especificando el nivel de acceso, modificadores opcionales como abstract o virtual, el valor de retorno, el nombre del método y cualquier parámetro del método. Los parámetros del método se encierran entre paréntesis y se separan por comas. Los paréntesis vacíos indican que el método no requiere parámetros.

En el siguiente ejemplo la clase contiene tres métodos:

```
abstract class Motorcycle
{
    // Anyone can call this.
    public void StartEngine() { /* Method statements here */ }

    // Only derived classes can call this.
    protected void AddGas(int gallons) { /* Method statements here */ }

    // Derived classes can override the base class implementation.
    public virtual int Drive(int miles, int speed) { /* Method statements here */ return 1; }

    // Derived classes must implement this.
    public abstract double GetTopSpeed();
}
```

Para llamar a un método con un objeto, después del nombre del objeto, se agrega un punto, el nombre del método y paréntesis. Los argumentos se enumeran entre los paréntesis y estarán separados por comas. Los métodos de la clase Motorcycle se pueden llamar como en el ejemplo siguiente:

```

class TestMotorcycle : Motorcycle
{
    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {
        TestMotorcycle moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine("My top speed is {0}", speed);
    }
}

```

Retorno de valores:

Los métodos pueden devolver un valor al autor de la llamada. Si el tipo de valor devuelto, el tipo especificado antes del nombre de método, no es void, el método debe devolver el valor mediante la utilización de la palabra clave return. Una instrucción con la palabra clave return seguida de un valor que coincide con el tipo de valor devuelto devolverá este valor al autor de la llamada del método.

La palabra clave return también detiene la ejecución del método si el tipo de valor devuelto es void. Una instrucción return sin un valor también es útil para detener la ejecución del método. Por otro lado, sin la palabra clave return, el método dejará de ejecutarse cuando alcance el final del bloque de código.

Por ejemplo, estos dos métodos utilizan la palabra clave return para devolver valores enteros:

```

class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2)
    { return number1 + number2; }

    public int SquareANumber(int number)
    { return number * number; }
}

```

Para utilizar un valor devuelto de un método, el método de llamada puede usar la llamada de método en cualquier lugar y puede hacerlo de distintas formas:

- Con un valor del mismo tipo.
- Con una variable que guarde un valor.
- Con la llamada a otro método que devuelva un valor.

Los dos siguientes ejemplos de código logran el mismo objetivo:

```
int result = obj.AddTwoNumbers(1, 2);  
result = obj.SquareANumber(result);  
// The result is 9.  
Console.WriteLine(result);
```

```
int result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));  
// The result is 9.  
Console.WriteLine(result);
```

Usar una variable local, en este caso, **result**, para almacenar un valor es opcional. Puede ser útil para la legibilidad del código, o puede ser necesaria si se debe almacenar el valor original del argumento para usarse en todo el ámbito del método.

2.5 – Constructores y Destructores: Declaración, uso y aplicaciones.

Constructor :

En programación orientada a objetos (POO), un constructor es un método especial dentro de una clase, que se llama automáticamente cada vez que se crea un objeto de esa clase y cuya misión es inicializar el objeto asignándole sus valores iniciales.

Características del método constructor:

1. Llamarse exactamente igual que la clase que lo define.
2. No puede retornar un valor.
3. Puede recibir parámetros.
4. Recomendable de acceso publico, para permitir sea utilizado fuera de la clase.
5. Puede sobrecargarse.

Tipos de constructores:

Constructor por defecto.

Un constructor por defecto es un constructor sin parámetros que no hace nada. Sin embargo será invocado cada vez que se construya un objeto sin especificar ningún argumento, en cuyo caso el objeto será iniciado con los valores predeterminados por el sistema (los atributos numéricos a ceros, los alfanuméricos a nulos, y las referencias a objetos a null).

Por ejemplo, un constructor por defecto para la clase SSuma quedaría así:

```
public SSuma () {}
```

Como se observa el constructor no posee ningún parámetro, ya que como no ha sido definido propiamente por el programador, el compilador lo hace por default.

Si se define un constructor con parámetros (definido explícitamente) el constructor por default se reemplaza por este.

Constructor con Parámetros.

Un constructor con Parámetros siempre debe especificar algún argumento, en cuyo caso el objeto será iniciado con los valores indicados por el usuario, y realizará también las acciones programadas al ejecutarse.

A continuación se creará un constructor explícito para una clase simple, utilizando una clase Persona con una serie de atributos definidos previamente.

```
class Persona
{
    //ATRIBUTOS
    private string nombre;
    private int edad;
    private string empleo;
    private double salario;

    //CONSTRUCTORES
    public Persona() {} //CONSTRUCTOR SIN PARÁMETROS
    public Persona(string nombre, int edad, string empleo, double
salario)
    {
        asignarNombre(nombre);
        asignarEdad(edad);
        asignarEmpleo(empleo);
        asignarSalario(salario);
    }
    //...
}
```

El constructor sin parámetros es reemplazado por el constructor explícito. En este ejemplo se puede observar que los constructores preferentemente se declaran públicos para que puedan ser invocados desde cualquier parte del programa.

Una línea como esta invocará al constructor sin parámetros:

```
Persona pers01 = new Persona(); // invoca al constructor Persona
```

El operador new crea un nuevo objeto, en este caso de la clase Persona, y a continuación se invoca al constructor de la clase para realizar las operaciones de iniciación que estén programadas.

A continuación en la siguiente línea de código, se invocará al constructor con parámetros, que se introducirán en el orden especificado en el constructor.

```
Persona pers02 = new Persona("Pepe Pérez", 30, "Programador", 25000);
```

Destructor :

Un destructor en algunos lenguajes de programación orientados a objetos es un método de una clase, que se llama justo antes de que una instancia de esa clase sea eliminada de la memoria. No todos los lenguajes de programación orientados a objetos suelen tener un destructor.

Es la contrapartida de un constructor que se ejecuta cuando se crea el objeto, se instancia y se le inicializa.

Un destructor en programación orientada a objetos es una función miembro especial llamadas automáticamente en la ejecución del programa, y por tanto no tienen por qué ser llamadas explícitamente por el programador.

Sus principales cometidos son:

- Liberar los recursos computacionales que el objeto de dicha clase haya adquirido en su tiempo de ejecución al expirar éste.
- Quitar los vínculos que pudiesen tener otros recursos u objetos con éste.

Los destructores son invocados automáticamente al alcanzar el flujo del programa el fin del ámbito en el que está declarado el objeto.

Características del método destructor:

1. Llamarle exactamente igual que la clase que lo define precedido de un símbolo de tilde (~).
2. No admite modificadores de acceso.
3. No permite recibir argumentos.
4. No puede sobrecargarse, solo puede tenerse 1 por clase.
5. No permiten heredarse.

Los destructores se invocan automáticamente y no se pueden invocar explícitamente. Una instancia se convierte en candidata para destrucción cuando ya ninguna parte de código puede utilizarla. La ejecución del destructor de la instancia puede ocurrir en cualquier momento una vez que la instancia se convierta en candidata para destrucción. Cuando se destruye una instancia, se llama a los destructores de su cadena de herencia en orden, de la más derivada a la menos derivada. Un destructor puede ejecutarse en cualquier subproceso.

El resultado de este ejemplo:

```
using System;

class A
{
    ~A() {
        Console.WriteLine("A's destructor");
    }
}

class B: A
{
    ~B() {
        Console.WriteLine("B's destructor");
    }
}

class Test
{
    static void Main() {
        B b = new B();
        b = null;
    }
}
```

Seria el siguiente:

```
B's destructor
A's destructor
```

Ya que en una cadena de herencia los destructores se llaman en orden, de la más derivada a la menos derivada, hasta llegar a la clase base.

2.6 – Sobrecarga de métodos.

En programación orientada a objetos la sobrecarga se refiere a la posibilidad de tener dos o más funciones con el mismo nombre pero funcionalidad diferente. Es decir, dos o más funciones con el mismo nombre pero que realizan acciones diferentes. El compilador usará una u otra dependiendo de los parámetros usados en la llamada al método.

La sobrecarga de métodos permite definir dos o más métodos con el mismo nombre, pero que difieren en cantidad o tipo de parámetros.

Esta característica del lenguaje nos facilita la implementación de algoritmos que cumplen la misma función pero que difieren en los parámetros que reciben para ejecutarse.

Ejemplo 1:

Implementar dos métodos que sumen los parámetros que reciben; dos enteros en el primer caso y que concatenen dos string en el segundo caso.

```
class Program {  
  
    public int Sumar(int x1, int x2)  
    {  
        int s = x1 + x2;  
  
        return s;  
    }  
  
    public string Sumar(string s1, string s2)  
    {  
        string s = s1 + s2;  
  
        return s;  
    }  
  
    static void Main(string[] args){  
  
        Program p = new Program();  
  
        Console.WriteLine("La suma de 5+10 es:" + p.Sumar(5, 10));  
    }  
}
```

```
Console.WriteLine("La concatenación de \"Juan\" y \" Carlos\" es "+p.Sumar("Juan", " Carlos"));
```

```
Console.ReadKey();
```

```
}
```

```
}
```

Como podemos ver definimos dos métodos llamados Sumar con dos parámetros cada uno (el primero con parámetros de tipo int y el segundo con parámetros de tipo string):

```
public int Sumar(int x1, int x2)
```

```
{
```

```
    int s = x1 + x2;
```

```
    return s;
```

```
}
```

```
public string Sumar(string s1, string s2)
```

```
{
```

```
    string s = s1 + s2;
```

```
    return s;
```

```
}
```

Cuando llamamos a los métodos el compilador sabe con cual enlazarlos según el tipo de datos enviados.

Si le enviamos dos enteros se llama el primer método:

```
Console.WriteLine("La suma de 5+10 es:" + p.Sumar(5, 10));
```

Si le enviamos dos string se llama el segundo método:

```
Console.WriteLine("La concatenación de \"Juan\" y \" Carlos\" es "+p.Sumar("Juan", " Carlos"));
```

Ejemplo 2:

Plantear una clase llamada Ventana que defina cuatro métodos sobrecargados que muestren un mensaje en la consola.

- El primero lo muestra donde se encuentra actualmente el cursor.
- El segundo lo muestra en una determinada columna y fila.
- El tercero lo muestra en una determinada columna, fila y con un color de letra.
- El cuarto lo muestra en una determinada columna, fila, con un color de letra más un color de fondo.

```
class Ventana
{
    public void Mostrar(string mensaje)
    {
        Console.Write(mensaje);
    }

    public void Mostrar(string mensaje, int columna, int fila)
    {
        Console.SetCursorPosition(columna, fila);
        Console.Write(mensaje);
    }

    public void Mostrar(string mensaje, int columna, int
fila, ConsoleColor colorletra)
    {
        Console.ForegroundColor = colorletra;
        Mostrar(mensaje, columna, fila);
    }

    public void Mostrar(string mensaje, int columna, int
fila, ConsoleColor colorletra, ConsoleColor colorfondo)
    {
        Console.ForegroundColor = colorletra;
        Console.BackgroundColor = colorfondo;
        Mostrar(mensaje, columna, fila);
    }
}
```

```

    {
        Console.BackgroundColor = colorfondo;
        Mostrar(mensaje, columna, fila, colorletra);
    }

    static void Main(string[] args)
    {
        Ventana v = new Ventana();
        v.Mostrar("Hola Mundo");
        v.Mostrar("Hola Mundo", 30, 10);
        v.Mostrar("Hola Mundo", 30, 12, ConsoleColor.Red);
        v.Mostrar("Hola Mundo", 30, 14, ConsoleColor.Red,
        ConsoleColor.Blue);
        Console.ReadKey();
    }
}

```

Como podemos observar hemos definido cuatro métodos llamados Mostrar que difieren en la cantidad de parámetros.

El primero recibe un string y lo muestra en la consola:

```

public void Mostrar(string mensaje)
{
    Console.Write(mensaje);
}

```

El segundo recibe tres parámetros con el mensaje, la columna y fila donde mostrarlo:

```

public void Mostrar(string mensaje, int columna, int fila)
{
    Console.SetCursorPosition(columna, fila);
}

```

```
        Console.Write(mensaje);  
    }  
}
```

Como podemos observar la clase Console tiene un método llamado `SetCursorPosition` que le pasamos la columna y fila donde queremos que se posicione el cursor previo a la salida de datos llamando al método `Write`.

El tercer método recibe cuatro parámetros y como podemos ver desde dentro de este método llamamos al método `Mostrar` que recibe tres parámetros:

```
    public void Mostrar(string mensaje, int columna, int fila, ConsoleColor colorletra)  
    {  
        Console.ForegroundColor = colorletra;  
        Mostrar(mensaje, columna, fila);  
    }  
}
```

Para cambiar el color de la letra de la Console debemos inicializar la propiedad `ForegroundColor`.

El cuarto método es similar al tercero pero con un quinto parámetro:

```
    public void Mostrar(string mensaje, int columna, int fila, ConsoleColor colorletra,  
        ConsoleColor colorfondo)  
    {  
        Console.BackgroundColor = colorfondo;  
        Mostrar(mensaje, columna, fila, colorletra);  
    }  
}
```

2.7 – Sobrecarga de operadores: Concepto y utilidad, operadores unarios y binarios.

Los lenguajes de programación, permiten sobrecargar operadores para utilizarlos en clases propias. Esto hace posible que utilizar un tipo de datos definido por el usuario parezca tan natural y lógico como utilizar un tipo de datos fundamental. Por ejemplo, podría crearse un nuevo tipo de datos denominado ComplexNumber para representar un número complejo y proporcionar métodos que realicen operaciones matemáticas en esos números mediante operadores aritméticos estándar, como utilizar el operador + para sumar dos números complejos.

Para sobrecargar un operador, se escribe un método especial con la siguiente sintaxis general:

```
public static <tipoDevuelto> operator <símbolo> (<operandos>)
{
    <cuerpo>
}
```

- El método debe ser publico y estático. Los modificadores pueden permutarse si se desea, lo que es importante es que siempre aparezcan en toda sobrecarga de operador.
- <tipoDevuelto> no puede ser **void**, pues por definición toda operación tiene un resultado, por lo que todo operador ha de devolver algo.
- Su nombre sera el <símbolo> del operador a redefinir precedido de la palabra reservada **operator**.
- En <operandos> se ha de incluir tantos parámetros como operandos pueda tomar el operador a redefinir, ya que cada uno representara a uno de sus operandos. Esto dependiendo de que sea unario o binario.
- En <cuerpo> se han de escribir las instrucciones a ejecutar cada vez que se aplique la operación cuyo operador es <símbolo> a los operandos de los tipos indicados en <operandos>.

Restricciones al sobrecargar operadores:

- Al menos uno de los operandos ha de ser del mismo tipo de dato del que sea miembro la sobrecarga del operador.
- No puede alterarse sus reglas de precedencia, asociatividad, ubicación y numero de operandos.

- No puede definirse nuevos operadores ni combinaciones de los ya existentes con nuevos significados.
- No todos los operadores incluidos en el lenguaje pueden redefinirse, pues muchos de ellos (**new**, **=**, etc) son básicos para el lenguaje y su sobrecarga es inviable, poco útil o dificultaría innecesariamente la legibilidad del código.

a) Sobrecarga de Operadores Unarios:

Los únicos operadores unarios redefinibles son: **!**, **+**, **-**, **~**, **++**, **--**, **true** y **false**, toda sobrecarga de un operador unario ha de tomar un único parámetro que ha de ser del mismo tipo que el tipo de dato al que pertenezca la sobrecarga.

Los operadores **++** y **--** siempre han de redefinirse de manera que el tipo de dato del objeto devuelto sea el mismo que el tipo de dato donde se definen.

Respecto a los operadores **true** y **false**, estos indican respectivamente, cuando se ha de considerar que un objeto representa el valor lógico verdadero y cuando se ha de considerar que representa el valor lógico falso, por lo que sus redefiniciones siempre han de devolver un objeto de tipo **bool** que indique dicha situación. Además, si se redefine uno es obligatorio redefinir también el otro, pues siempre es posible usar indistintamente uno u otro para determinar el valor lógico que un objeto represente.

Por ejemplo, a continuación se muestra cómo se redefiniría el significado de estos operadores para los objetos de la clase **ComplexNumber** : Si sabemos que un número complejo tiene una parte real y una imaginaria.

Tratemos de simular los números complejos a través de la clase **ComplexNumber** en C#:

```
public class ComplexNumber
{
    public int real;
    public int imaginary;

    //Constructor de la clase
    public ComplexNumber(int real, int imaginary)
    {
        this.real = real;
        this.imaginary = imaginary;
    }

    // Se declara cual operador será sobrecargado, en este caso ++
    (incremento)
    // y el resultado será de tipo ComplexNumber también
    public static ComplexNumber operator ++(ComplexNumber c1)
    {

```

```

        return new ComplexNumber(c1.real + 1, c1.imaginary + 1);
    }
    // Redefinición del operador true
    public static bool operator true(ComplexNumber c1)
    {
        return (c1.real != 0 || c1.imaginary != 0);
    }
    //Redefinición del operador false, ya que definimos antes true
    public static bool operator false(ComplexNumber c1)
    {
        return (c1.real == 0 && c1.imaginary == 0);
    }
    // Sobrecargamos también el método ToString() para mostrar el
    resultado de
    // forma personalizada en pantalla
    public override string ToString()
    {
        return(String.Format("{0} + {1}i", real, imaginary));
    }

    public static void Main()
    {
        ComplexNumber c1 = new ComplexNumber(2,3);

        // Incremento del ComplexNumber a traves del sobrecargar el operador
        ++
        c1++;

        // Se muestra el resultado usando el método ToString() sobrecargado
        Console.WriteLine("El incremento del número complejo es: {0}",c1);

        // Los operadores true y false no pueden usarse directamente el el
        código fuente.
        // Con las redefiniciones anteriores, se utilizarían las lineas
        siguientes para mostrar
        // por pantalla el mensaje "Es Verdadero"
        if(c1)
            Console.WriteLine("El valor del número complejo {0} Es
            Verdadero",c1);
    }
}

```

b) Sobrecarga de Operadores Binarios:

Algunos de los operadores binarios redefinibles son: +, -, *, /, %, &, |, <<, >>, ==, !=, >, <, >= y <=.

Toda sobrecarga que se haga de ellos ha de tomar dos parámetros tales que al menos uno sea del mismo tipo que el tipo de dato del que es miembro la sobrecarga.

Hay que tener en cuenta que aquellos de estos operadores que tengan complementario, siempre han de redefinirse junto con este. Es decir, siempre que se redefina en un tipo el operador > también ha de redefinirse en él el operador <, siempre que se redefina >= ha de redefinirse <=, y siempre que se redefina == ha de redefinirse !=.

El ejemplo en C# quedaría de la siguiente manera:

```
public class ComplexNumber
{
    public int real;
    public int imaginary;

    //Constructor de la clase
    public ComplexNumber(int real, int imaginary)
    {
        this.real = real;
        this.imaginary = imaginary;
    }

    // Se declara cual operador será sobrecargado, en este caso + (suma)
    // Recibe 2 objetos de tipo ComplexNumber y el resultado será de tipo
    ComplexNumber
    public static ComplexNumber operator +(ComplexNumber c1, ComplexNumber
c2)
    {
        return new ComplexNumber(c1.real + c2.real, c1.imaginary +
c2.imaginary);
    }

    // Se redefine el operador ==, note que se puede usar dentro del
    método
    // el operador == porque se esta comparando int con int
    public static bool operator ==(ComplexNumber c1, ComplexNumber c2)
    {
        return c1.real == c2.real && c1.imaginary == c2.imaginary;
    }

    //Se redefine el operador !=, ya que se definió antes el operador ==
    public static bool operator !=(ComplexNumber c1, ComplexNumber c2)
    {
        return !(c1==c2);
    }

    // Sobrecargamos también el método ToString() para mostrar el resultado
    de
    // forma personalizada en pantalla

    public override string ToString()
    {
        return (String.Format("{0} + {1}i", real, imaginary));
    }

    public static void Main()
    {
        // Creación y asignación de valores a los objetos
        ComplexNumber c1 = new ComplexNumber(2,3);
        ComplexNumber c2 = new ComplexNumber(4,5);
        ComplexNumber c3 = new ComplexNumber(1,1);
        ComplexNumber c4 = new ComplexNumber(1,1);
    }
}
```

```

        // Suma de los 2 números complejos a través de la sobrecarga del
        operador "+"
        ComplexNumber suma = c1 + c2;

        // Mostramos los números y la suma usando el método ToString()
        sobrecargado
        Console.WriteLine("Primer número complejo: {0}",c1);
        Console.WriteLine("Segundo número complejo: {0}",c2);
        Console.WriteLine("La Suma de los dos números es: {0}",suma);

        // Mostramos la sobrecarga de los operadores == y !=
        Console.WriteLine(c3.Equals(c4));
    }
}

```

Resultados:

El incremento del numero complejo es: 3 + 4i

El valor del numero complejo 3 + 4i Es Verdadero

Primer numero complejo: 2 + 3i

Segundo numero complejo: 4 + 5i

La Suma de los dos números es: 6 + 8i

False

REFERENCIAS BIBLIOGRAFICAS UNIDAD 2

Declarar clases <https://msdn.microsoft.com/es-mx/library/x9afc042.aspx>

Atributos <http://www.sc.ehu.es/sbweb/fisica/cursoJava/fundamentos/atributos>

Método [https://es.wikipedia.org/wiki/Método_\(informática\)](https://es.wikipedia.org/wiki/Método_(informática))

Encapsulamiento [https://es.wikipedia.org/wiki/Encapsulamiento_\(informática\)](https://es.wikipedia.org/wiki/Encapsulamiento_(informática))
<https://algonzalezpoo.wordpress.com/encapsulamiento/>
http://www.ciberaula.com/articulo/encapsulacion_java

Instanciación de una clase personales.upv.es/rmartin/cursoJava/Java/OO/Instanciacion.htm

Referencia al objeto actual

p-o-o-itsav.blogspot.com/2012/02/23-referencia-al-objeto-actual.html

<https://sites.google.com/site/.../23-referencia-al-objeto-actual>

Métodos/declaración <https://palomaitor.wordpress.com/2011/06/08/unidad-2/>

Métodos/mensajes https://es.wikipedia.org/wiki/Paso_de_mensajes

Métodos/parámetros <https://msdn.microsoft.com/es-es/library/ms173114.aspx>

Constructor/destructor

gl-e-p-n-programacion-ii.blogspot.com/2010/03/constructor-y-destructor.html

[https://es.wikipedia.org/wiki/Constructor_\(inform%C3%A1tica\)](https://es.wikipedia.org/wiki/Constructor_(inform%C3%A1tica))

[https://es.wikipedia.org/wiki/Destructor_\(inform%C3%A1tica\)](https://es.wikipedia.org/wiki/Destructor_(inform%C3%A1tica))

[https://msdn.microsoft.com/es-s/library/aa645613\(v=vs.71\).aspx](https://msdn.microsoft.com/es-s/library/aa645613(v=vs.71).aspx)

Sobrecarga de métodos

<https://www.tutorialesprogramacionya.com/csharpya/detalleconcepto.php?codigo=189&inicio=60>

Sobrecarga de operadores

[https://msdn.microsoft.com/es-mx/library/ms228498\(v=vs.90\).aspx](https://msdn.microsoft.com/es-mx/library/ms228498(v=vs.90).aspx)

<http://www.puntopeek.com/codigos-c/sobrecarga-de-operadores-en-c-ejemplos/>

UNIDAD III – Herencia.

3.1 – Definición: clase base, clase derivada.

El mecanismo de herencia es uno de los pilares fundamentales en los que se basa la programación orientada a objetos, que permite definir nuevas clases a partir de otras ya definidas de modo que si en la definición de una clase indicamos que deriva de otra, entonces la primera la que se le suele llamar clase derivada (hija) será tratada por el compilador automáticamente como si su definición incluyese la definición de la segunda a la que se le suele llamar clase base (padre).

A la clase antecesora se le denomina **clase base** y esta hereda atributos y comportamientos a una clase sucesora la cual se llama **clase derivada**, a la que se le pueden agregar nuevos atributos y comportamientos.

A su vez, la clase derivada puede heredar sus atributos y comportamientos a otra nueva clase, de manera tal que se puede establecer una jerarquía similar a la que se utiliza en la clasificación taxonómica de las especies biológicas.

Las clases que derivan de otras se definen usando la siguiente sintaxis:

```
class <ClaseDerivada> : <ClaseBase>
{
    <miembrosDerivada>...
}
```

A los miembros definidos en <miembrosDerivada> se les añaden los que hubiéramos definido en la clase base.

Por **ejemplo**, a partir de la clase Persona puede crearse una clase Trabajador como sigue:

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

    //constructor clase derivada
    public Trabajador(string nombre, int edad, string nip, float sueldo)
    :base(nombre, edad, nip)
    {
        Sueldo = sueldo;
    }
}
```

Los objetos de esta clase Trabajador contarán con los mismos miembros que los objetos de la clase Persona y además incorporarán un nuevo campo llamado Sueldo que almacenará el dinero que percibe cada trabajador. Note además que a la hora de escribir el constructor de esta clase ha sido necesario escribirlo con una sintaxis especial

consistente en preceder la llave de apertura del cuerpo del método constructor de una estructura de la forma:

: base(<parametrosBase>)

A esta estructura se le llama inicializador base y se utiliza para indicar cuando deseamos inicializar los campos heredados de la clase padre. No es más que una llamada al constructor de la clase base con los parámetros adecuados. Si no se incluye el compilador considerará por defecto que vale: base (), lo que será incorrecto para este ejemplo y que provocaría un error ya que la clase base Persona carece de constructor sin parámetros.

El **ejemplo** de una forma más completa, que pone de manifiesto como funciona la herencia es el siguiente:

```
class Persona //Clase base
{
    public string Nombre; // Campo de objeto Persona que almacena su nombre
    public int Edad; // Campo de objeto Persona que almacena su edad
    public string NIP; // Campo de objeto Persona que almacena su NIP

    // Constructor de Persona
    public Persona (string nombre, int edad, string nip)
    {
        Nombre = nombre;
        Edad = edad;
        NIP = nip;
    }

    public void Cumpleaños( ) // Método que incrementa en uno la Edad
    {
        Edad++;
    }
} //fin clase base

class Trabajador:Persona //Clase derivada
{
    public float Sueldo; //Campo de objeto Trabajador que almacena su sueldo

    //Constructor de Trabajador
    public Trabajador(string nombre, int edad, string nip, float sueldo)
    :base(nombre, edad, nip) //Inicializa Trabajador en base al constructor
    de la clase Persona
    {
        Sueldo = sueldo;
    }
} //fin clase derivada

public static void Main(string[] args)
{
    //Creación del objeto Trabajador
    Trabajador t1 = new Trabajador("Josán", 22, "77588260-Z", 10000.00f);
    Console.WriteLine("Nombre= "+t1.Nombre);
    Console.WriteLine("Edad= "+t1.Edad);
    Console.WriteLine("NIP= "+t1.NIP);
    Console.WriteLine("Sueldo= "+t1.Sueldo);
}
```

Salida:
Nombre = Josan
Edad= 22
NIP= 77588260-Z
Sueldo= 10000.00

Note que ha sido necesario prefijar la definición de los miembros de Persona de la palabra reservada public. Esto se debe a que por defecto los miembros de un tipo solo son accesibles desde dentro de la definición de dicho tipo, e incluyendo public conseguimos que sean accesibles desde el método Main() definido en Trabajador.

3.2 – Clasificación: herencia simple, herencia múltiple.

Existen dos tipos de herencia:

1. Herencia simple y
2. Herencia múltiple.

En la **herencia simple**, una clase sucesora hereda de **una sola clase antecesora directa** (llamada **clase base**), como se muestra en la figura 3.2.1.

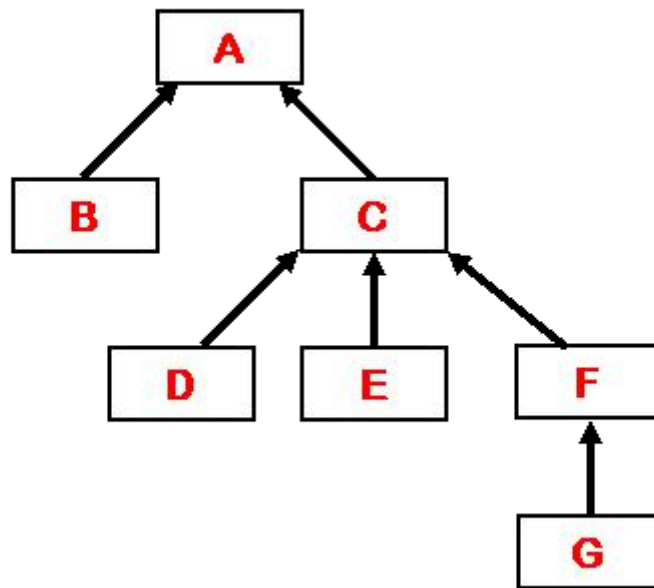


Figura 3.2.1.- Ejemplo de herencia simple

Las flechas en la figura 3.2.1 representan la relación "es un" o "es una", por lo que deben dirigirse **desde la clase sucesora hacia la antecesora**.

Herencia simple: Un objeto puede extender las características de otro objeto y de ningún otro, es decir, solo puede tener un padre.

Por ejemplo los lenguajes Java, Ada y C# sólo admiten herencia simple.

En la **herencia múltiple**, al menos una clase sucesora hereda de **más de una clase antecesora directa**, como se muestra en la figura 3.2.2.

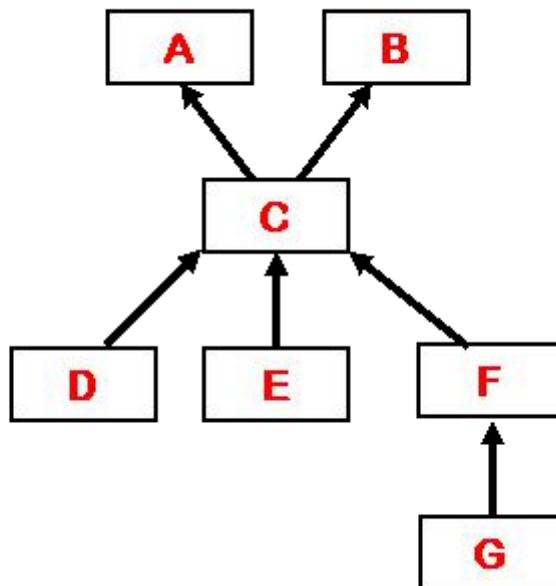


Figura 3.2.2.- Ejemplo de herencia múltiple

En este caso, la clase **C** tiene dos clases antecesoras directas (**A** y **B**)

Herencia múltiple: Un objeto puede extender las características de uno o más objetos, es decir, puede tener varios padres.

Por ejemplo el lenguaje C++ admite la herencia múltiple.

3.3 – Reutilización de miembros heredados.

La reutilización de código se refiere al comportamiento y a las técnicas que garantizan que una parte o la totalidad de un programa informático existente se puedan emplear en la construcción de otro programa. De esta forma se aprovecha el trabajo anterior, se economiza tiempo, y se reduce la redundancia.

La manera más fácil de reutilizar código es copiarlo total o parcialmente desde el programa antiguo al programa en desarrollo. Pero es trabajoso mantener múltiples copias del mismo código, por lo que en general se elimina la redundancia dejando el código reusable en un único lugar, y llamándolo desde los diferentes programas. Este proceso se conoce como abstracción. La abstracción puede verse claramente en las bibliotecas de software, en las que se agrupan varias operaciones comunes a cierto dominio para facilitar el desarrollo de programas nuevos. Hay bibliotecas para convertir información entre diferentes formatos conocidos, acceder a dispositivos de almacenamiento externos, proporcionar una interfaz con otros programas, manipular información de manera conocida (como números, fechas, o cadenas de texto).

Para que el código existente se pueda reutilizar, debe definir alguna forma de comunicación o interfaz. Esto se puede dar por llamadas a una subrutina, a un objeto, o a una clase.

Como se ha comentado anteriormente la clase descendiente puede añadir sus propios atributos y métodos pero también puede sustituir u ocultar los heredados. En concreto:

1. Se puede declarar un nuevo atributo con el mismo identificador que uno heredado, quedando este atributo oculto. Esta técnica no es recomendable.
2. Se puede declarar un nuevo método de instancia con la misma cabecera que el de la clase ascendiente, lo que supone su sobreescritura. Por lo tanto, la sobreescritura o redefinición consiste en que métodos adicionales declarados en la clase descendiente con el mismo nombre, tipo de dato devuelto y número y tipo de parámetros sustituyen a los heredados.
3. Se puede declarar un nuevo método de clase con la misma cabecera que el de la clase ascendiente, lo que hace que éste quede oculto. Por lo tanto, los métodos de clase o estáticos (declarados como static) no pueden ser redefinidos.
4. Un método declarado con el modificador final tampoco puede ser redefinido por una clase derivada.
5. Se puede declarar un constructor de la subclase que llame al de la superclase de forma implícita o de mediante la palabra reservada super.
6. En general puede accederse a los métodos de la clase ascendiente que han sido redefinidos empleando la palabra reservada super delante del identificador del método. Este mecanismo sólo permite acceder al método perteneciente a la clase en el nivel inmediatamente superior de la jerarquía de clases.

Para reutilizar el código creamos nuevas clases pero, en lugar de partir de cero, partimos de clases, relacionadas con nuestra clase, que han sido ya creadas y depuradas. El truco está en usar las clases sin ensuciar el código existente.

Una forma de hacer esto es crear objetos de nuestras clases existentes dentro de la nueva clase. Esto se conoce como composición porque la nueva clase está compuesta de objetos de clases existentes. Estamos reutilizando la funcionalidad del código, y no la forma.

Otra forma es crear una nueva clase como un tipo de una clase ya existente. Tomamos la forma de la clase existente y añadimos código a la nueva, sin modificar la clase existente. Esta forma de crear nuevos objetos se llamada herencia, y lo que hacemos es extender la clase en la que nos basamos para crear la nueva.

Composición:

La composición consiste en poner manejadores de objetos dentro de nuestra clase, estos manejadores de objetos no serán otra cosa que instancias de las clases en las que nos estamos basando para crear la nueva clase.

Recordemos que la forma para determinar cuando usar composición es cuando podemos decir que nuestra nueva clase “tiene un” elemento de otro tipo de objetos, por ejemplo un cronómetro tiene: horas, minutos y segundos, es decir que una clase Cronometro esta compuesta por otras clases llamadas: Horas, Minutos y Segundos.

Veamos como seria esta clase ejemplo:

```
public class Cronometro
{
    Horas h;
    Minutos m;
    Segundos s;
    string cadena;
    int seg,min,hor;

    public Cronometro()
    {
        seg=0;
        min=0;
        hor=0;
        h = new Horas();
        m = new Minutos();
        s = new Segundos();
        cadena = new string("0 : 0 : 0");
    }

    public string avanzar()
    {
        seg = s.forward();
        if(seg==0)
        { min=m.forward();
            if(min==0)
            { hor=h.forward(); }
        }
        cadena = hor + " : " + min + " : " + seg;
        return cadena;
    }

    public string reset()
    {
        seg = s.reset();
        min = m.reset();
    }
}
```

```

        hor = h.reset();
        cadena = hor + " : " + min + " : " + seg;
        return cadena;
    }
}

```

Nuestra clase Cronometro está compuesta entre otras cosas por objetos del tipo Horas, Minutos y Segundos y a través del constructor de nuestra clase creamos las instancias de cada una de ellas.

Herencia:

En java cuando heredamos, estamos diciendo "Esta nueva clase es como esa clase antigua", por ejemplo es decir que la clase Horas "es una" UnidadDeTiempo. Afirmamos esto en el código dando el nombre de la clase como siempre pero, antes de la apertura del límite cuerpo de clase, pondremos la palabra clave "extends" seguida por el nombre de la clase base. Cuando hagamos esto, obtendremos automáticamente todos los datos miembros y métodos de la clase base.

Primero veamos como seria la clase ejemplo UnidadDeTiempo:

```

public class UnidadDeTiempo
{
    int valor;
    int tope;

    public int forward()
    {
        if(valor == tope)
            valor=0;
        else
            valor++;
        return valor;
    }

    public int reset()
    {
        valor=0;
        return valor;
    }
}

```

Y nuestra clase Horas:

```

public class Horas extends UnidadDeTiempo
{
    public Horas()
    {
        this.valor=0;
        this.tope=23;
    }
}

```

De esta manera sin necesidad de tener que escribir nuevamente todo el código de UnidadDeTiempo lo tenemos disponible en la clase Horas.

El código de nuestras clases hijas no tiene porque limitarse solo al código heredado, de hecho casi siempre la herencia se hace para extender la funcionalidad de las clases heredadas añadiendo nuevos métodos y atributos.

La composición y la herencia:

Tanto la composición como la herencia permiten poner sub-objetos dentro de tu nueva clase. Podríamos preguntarnos cuál es la diferencia entre los dos, y cuándo elegir uno en lugar del otro. La composición es generalmente usada cuando deseamos las características de una clase existente dentro de una nueva clase, pero no su interfaz. Es decir, ponemos un para poder usarlo para implementar características de nuestra nueva clase, pero el usuario de esa nueva clase verá el interfaz que hemos definido en lugar del interfaz del objeto insertado.

Los objetos miembros usan la implementación ocultándose a sí mismos, por lo que esto es una cosa segura a hacer y, cuando el usuario sabe que estamos uniendo un conjunto de partes, hace que el interfaz sea más fácil de entender.

Cuando heredamos, estamos tomando una clase existente y creando una versión especial de esa clase. En general, esto significa que estamos tomando una clase de propósito general, especializándola para un caso o necesidad particular.

La relación “es-un” viene expresada por la herencia, y la relación “tiene-un” viene expresada por la composición.

Se puede hacer que una clase derivada cambie la implementación de un método de una clase base, manteniendo el nombre del método.

A esta operación de reimplementar un método de una clase base en una clase derivada se le conoce como reemplazar (**override**) el método de la clase base.

Los métodos de la clase base que usan la palabra clave **virtual** reciben el nombre de métodos virtuales y los de la clase derivada que usan la palabra clave **override** reciben el nombre de métodos de reemplazo.

Para lograr el reemplazo de métodos deben cumplirse dos requisitos:

1. El método de la clase base debe declararse con la palabra clave **virtual**.
2. El método de la clase derivada debe declararse con la palabra clave **override**.

La implementación de un método no virtual es invariable. La implementación es la misma tanto si se invoca un método en una instancia de la clase en la que se declaró o en una instancia de una clase derivada. En cambio, la implementación de un método virtual se puede sustituir por clases derivadas. El proceso de sustitución de la implementación de un método virtual heredado es conocido como reemplazamiento del método.

Cuando se invoca a un método virtual, el tipo del objeto se comprueba en tiempo de ejecución para ver si existe un miembro de reemplazo y se realiza una llamada al miembro de reemplazo que está en la clase de mayor derivación, el cual puede ser el miembro original, para ver si no existe ninguna clase derivada que haya reemplazado el miembro.

El modificador **virtual** no se puede utilizar con los siguientes modificadores:

- **static**
- **abstract**
- **override**

Las propiedades virtuales funcionan como los métodos abstractos, salvo en lo que se refiere a las diferencias en la sintaxis de las declaraciones e invocaciones.

- Es incorrecto utilizar el modificador **virtual** en una propiedad estática.
- Una propiedad virtual heredada se puede reemplazar en una clase derivada si se incluye una declaración de propiedad que use el modificador **override**.

Las clases que tienen métodos virtuales son instanciables, esto es, se pueden construir objetos de esas clases.

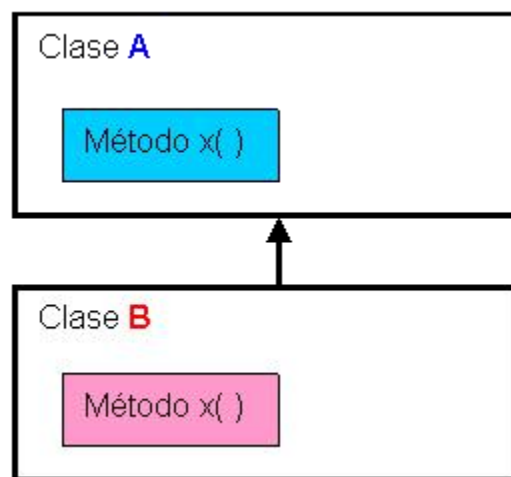
Ejemplo del reemplazamiento de métodos en clases derivadas.

```
class Program
{
    class A //clase base
    {
        public virtual void X( )
        {
            Console.WriteLine( "Método X de A" ) ;
        }
    }

    class B : A //clase derivada
    {
        public override void X( )
        {
            Console.WriteLine( "Método X de B" ) ;
        }
    }

    public static void Main(string[] args)
    {
        A objeto1 = new A( ) ;
        B objeto2 = new B( ) ;
        objeto1.X( ) ;
        objeto2.X( ) ;
    }
}
```

Salida:
Método X de A
Método X de B



3.4 – Referencia al objeto de la clase base.

La funcionalidad de una clase existente se puede extender al crear una nueva clase que se deriva de ella. La clase derivada hereda las propiedades de la clase base y es posible agregar o reemplazar métodos y propiedades según sea necesario.

En C#, el operador :, que equivale a extends e implements en Java, define la herencia e implementación de interfaces. La clase base siempre debe estar en el extremo izquierdo en la declaración de clase.

Como Java, C# no admite herencia múltiple, lo que significa que las clases no pueden heredar más de una clase. Sin embargo, se pueden utilizar interfaces para ese propósito, de la misma manera que en Java.

El código siguiente define una clase denominada CoOrds con dos variables miembro privadas x e y que representan la posición del punto. Se tiene acceso a estas variables mediante propiedades denominadas X e Y, respectivamente:

```
public class CoOrds //clase base
{
    private int x, y; //campos clase base

    public CoOrds() // constructor clase base
    {
        x = 0;
        y = 0;
    }

    public int X //propiedad
    {
        get { return x; }
        set { x = value; }
    }

    public int Y //propiedad
    {
        get { return y; }
        set { y = value; }
    }
}
```

Una nueva clase, denominada ColorCoOrds, hereda todos los campos y métodos de la clase base; y a la cual se pueden agregar nuevos campos y métodos para proporcionar características adicionales en la clase derivada, según sea necesario.

En este ejemplo, se agrega un miembro privado y descriptores de acceso para agregar color a la clase:

```

public class ColorCoOrds : CoOrds //clase derivada
{
    private System.Drawing.Color screenColor; //campo clase derivada

    public ColorCoOrds() // constructor clase derivada
    {
        screenColor = System.Drawing.Color.Red;
    }

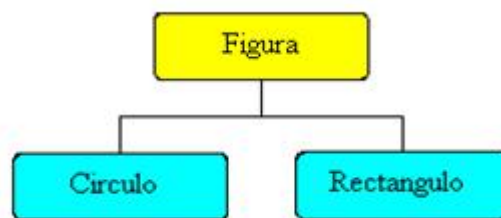
    public System.Drawing.Color ScreenColor //propiedad
    {
        get { return screenColor; }
        set { screenColor = value; }
    }
}

```

El constructor de la clase derivada llama implícitamente al constructor de la clase base o la superclase, en terminología de Java. En caso de herencia, se llama a todos los constructores de clase base antes que a los constructores de la clase derivada en el orden en que las clases aparecen en la jerarquía de clases.

La jerarquía de clases que describen las figuras planas

Consideremos las figuras planas cerradas como el rectángulo, y el círculo. Tales figuras comparten características comunes como es la posición de la figura, de su centro, y el Área de la figura, aunque el procedimiento para calcular dicha Área sea completamente distinto para cada una. Podemos por tanto, diseñar una jerarquía de clases, tal que la clase base denominada Figura, tenga las características comunes y cada clase derivada las específicas. La relación jerárquica se muestra en la siguiente figura



La clase Figura es la que contiene las características comunes a dichas figuras concretas por tanto, no tiene forma ni tiene Área. Esto lo expresamos declarando Figura como una clase abstracta, declarando la función miembro área abstract.

Las clases abstractas solamente se pueden usar como clases base para otras clases. No se pueden crear objetos pertenecientes a una clase abstracta. Sin embargo, se pueden declarar variables de dichas clases.

a) La clase Figura:

La definición de la clase abstracta Figura, contiene la posición x e y de la figura particular, de su centro, y la función área, que se va a definir en las clases derivadas para calcular el Área de cada figura en particular.

```
public abstract class Figura
{
    protected int x;
    protected int y;

    public Figura(int x, int y)
    {
        this.x=x;
        this.y=y;
    }
    public abstract double area();
}
```

b) La clase Rectángulo:

Las clases derivadas heredan los miembros dato x e y de la clase base, y definen la función área, declarada abstract en la clase base Figura, ya que cada figura particular tiene una formula distinta para calcular su Área. Por ejemplo, la clase derivada Rectángulo, tiene como datos, aparte de su posición (x, y) en el plano, sus dimensiones, es decir, su anchura (ancho) y altura (alto).

```
class Rectangulo extends Figura
{
    protected double ancho, alto;

    public Rectangulo(int x, int y, double ancho, double alto)
    {
        super(x,y);
        this.ancho=ancho;
        this.alto=alto;
    }

    public double area()
    {
        return ancho*alto;
    }
}
```

La primera sentencia en el constructor de la clase derivada es una llamada al constructor de la clase base, para ello se emplea la palabra reservada super. El constructor de la clase derivada llama al constructor de la clase base y le pasa las coordenadas del punto x e y. Después inicializa sus miembros dato ancho y alto.

En la definición de la función área, se calcula el Área del rectángulo como producto de la anchura por la altura, y se devuelve el resultado

c) La clase Círculo:

```
class Circulo extends Figura
{
    protected double radio;

    public Circulo(int x, int y, double radio)
    {
        super(x,y);
        this.radio=radio;
    }

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

Como vemos, la primera sentencia en el constructor de la clase derivada es una llamada al constructor de la clase base empleando la palabra reservada super. Posteriormente, se inicializa el miembro dato radio, de la clase derivada Círculo.

En la definición de la función área, se calcula el Área del círculo mediante la conocida formula πr^2 , o bien $\pi \cdot r \cdot r$. La constante Math.PI es una aproximación decimal del número irracional π .

Uso de la jerarquía de clases

Creemos un objeto c de la clase Círculo situado en el punto (0, 0) y de 5.5 unidades de radio. Calculamos y mostramos el valor de su Área.

```
Circulo c=new Circulo(0, 0, 5.5);
System.out.println("Area del círculo "+c.area());
```

Creemos un objeto r de la clase Rectángulo situado en el punto (0, 0) y de dimensiones 5.5 de anchura y 2 unidades de largo. Calculamos y mostramos el valor de su Área.

```
Rectangulo r=new Rectangulo(0, 0, 5.5, 2.0);
System.out.println("Area del rectángulo "+r.area());
```

Veamos ahora, una forma alternativa, guardamos el valor devuelto por new al crear objetos de las clases derivadas en una variable f del tipo Figura (clase base).

```
Figura f=new Circulo(0, 0, 5.5);
System.out.println("Area del círculo "+f.area());
f=new Rectangulo(0, 0, 5.5, 2.0);
System.out.println("Area del rectángulo "+f.area());
```

3.5 – Constructores y destructores en clases derivadas.

a) Constructores en clases derivadas.

Al instanciar objetos de clases derivadas se inicia una cadena de invocaciones a constructores en las cuales el constructor de la clase derivada, antes de realizar sus propias tareas, invoca (ya sea implícita o explícitamente) al constructor de su clase base.

Similarmente, si la clase base fue derivada de otra clase, el constructor de la clase base debe invocar al constructor de la clase ubicada en el siguiente nivel superior de la jerarquía, y así sucesivamente.

El último constructor invocado en la cadena es el constructor de la clase **Object**, cuyo cuerpo se ejecuta primero. El cuerpo del constructor de la clase derivada se ejecuta al final.

El constructor de cada clase base inicializa las variables de instancia que el objeto de la clase derivada hereda.

b) Destructores en clases derivadas.

Cuando se remueve de la memoria un objeto de una clase derivada, el recolector de basura invoca al destructor del objeto.

Esto inicia una cadena de invocaciones a destructores, en donde el destructor de la clase derivada y los destructores de las clases bases directas e indirectas se ejecutan en orden inverso al que se ejecutaron los constructores, esto es, primero se ejecuta el destructor de la clase derivada y al final se ejecuta el destructor de la clase base ubicada en el nivel superior de la jerarquía.

La ejecución de los destructores debe liberar todos los recursos que el objeto adquirió, antes de que el recolector de basura reclame la memoria de ese objeto.

Cuando el recolector de basura invoca al destructor de un objeto de una clase derivada, ese destructor realiza su tarea y después invoca al destructor de la clase base.

El proceso se repite hasta que se invoca al destructor de la clase **Object** al final.

Ejemplo de: **Destructores en clases derivadas.**

```
namespace ConsoleApplication1
{
    class Program
    {
        class Animal //clase base
        {
            //destructor
            ~ Animal( )
            {
                Console.WriteLine("Muere mi parte Animal ...");
            }
        }

        class Mamífero : Animal //clase derivada de animal
        {
            //destructor
            ~ Mamífero( )
            {
                Console.WriteLine("Muere mi parte Mamífero ...");
            }
        }

        class Perro : Mamífero //clase derivada de mamífero
        {
            //destructor
            ~ Perro( )
            {
                Console.WriteLine("Muere mi parte Perro ...");
            }
        }

        public static void Main(string[] args)
        {
            //creación del objeto
            Perro Fido = new Perro ( );
        }
    }
}
```

Salida:

Muere mi parte Perro...

Muere mi parte Mamífero...

Muere mi parte Animal...

3.6 – Redefinición de métodos en clases derivadas.

Los lenguajes de programación permiten redefinir miembros (campos y métodos) de la clase base en las clases derivadas, pero el compilador emite una advertencia cuando detecta una redefinición.

Una advertencia (**warning**) es un mensaje del compilador acerca de un posible problema. Sin embargo, en este caso sí se genera código ejecutable (a diferencia del mensaje de error).

a) Redefinición de campos.

El siguiente ejemplo muestra cómo reutilizar los identificadores de los campos de la clase base en una clase derivada.

Ejemplo:

```
class Program
{
    class Punto //clase base
    {
        public int x;
        public int y;
    }

    class Punto3D : Punto //clase derivada
    {
        public int x ;
        public int y ;
        public int z ;
    }

    public static void Main(string[] args )
    {
        Punto a = new Punto( );
        Punto3D b = new Punto3D( );
        a.x = 100 ;
        a.y = 200 ;
        b.x = 300 ;
        b.y = 400 ;
        b.z = 500 ;
    }
}
```

El compilador envía dos advertencias al generar el código ejecutable:

..."La palabra clave **new** es necesaria en 'Punto3D.x' porque oculta el miembro heredado 'Punto.x' "...

..."La palabra clave **new** es necesaria en 'Punto3D.y' porque oculta el miembro heredado 'Punto.y' "...

La palabra clave **new** tiene dos implementaciones diferentes: como operador y como modificador.

- El **operador new** se usa para crear objetos (p.ej. **Cuenta c = new Cuenta();**)
- El **modificador new** se usa para ocultar un miembro heredado de un miembro de clase base (p.ej. **new public int x;**)

Ejemplo de la redefinición de campos en clases derivadas, sin emitir advertencias por parte del compilador:

```
class Program
{
    class Punto //clase base
    {
        public int x;
        public int y;
    }

    class Punto3D : Punto //clase derivada
    {
        new public int x ;
        new public int y ;
        public int z ;
    }

    public static void Main(string[] args )
    {
        Punto a = new Punto( );
        Punto3D b = new Punto3D( );
        a.x = 100 ;
        a.y = 200 ;
        b.x = 300 ;
        b.y = 400 ;
        b.z = 500 ;
    }
}
```

b) Redefinición de métodos.

Ejemplo utilizando **new**:

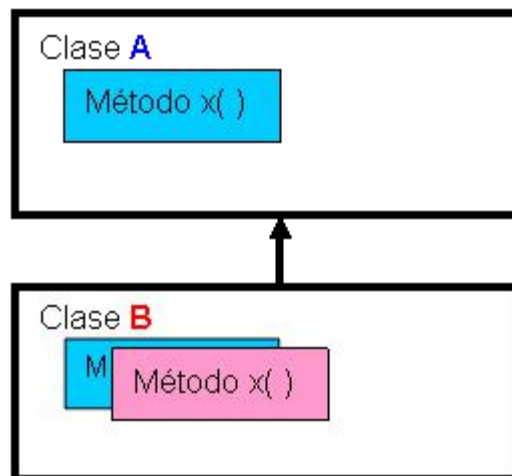
```
class Program
{
    class A //clase base
    {
        public void X( )
        {
            Console.WriteLine( "Método X de A" ) ;
        }
    }

    class B : A //clase derivada
    {
        new public void X( )
        {
            Console.WriteLine( "Método X de B" ) ;
        }
    }

    public static void Main(string[] args )
    {
        A objeto1 = new A( ) ;
        B objeto2 = new B( ) ;
        objeto1.X( ) ;
        objeto2.X( ) ;
    }
}
```

Salida:

Método X de A
Método X de B



REFERENCIAS BIBLIOGRAFICAS UNIDAD 3

Definición: clase base, clase derivada.

<http://www.sc.ehu.es/sbweb/fisica/cursoJava/fundamentos/herencia/herencia.htm>

[https://msdn.microsoft.com/es-mx/library/ms228387\(v=vs.90\).aspx](https://msdn.microsoft.com/es-mx/library/ms228387(v=vs.90).aspx)

Clasificación: herencia simple, herencia múltiple.

<http://pooitzo.blogspot.mx/clasificacion-herencia-simple-y.html>

<http://cesargenaro.blogspot.mx/clasificacion-de-herencia.html>

Reutilización de miembros heredados

<https://sites.google.com/site/wwwprogramacionorientadacom/33-reutilizacion-de-miembros-heredados>

<http://apdaza-poo.blogspot.mx/reutilizacin-del-codigo-y-relaciones.html>

Referencia al objeto de la clase base

<http://fjglez90.blogspot.mx/referencia-al-objeto-de-la-clase.html>

<http://invfinalisc.blogspot.mx/referencia-al-objeto-de-la-clase.html>

Constructores y destructores en clases derivadas.

<http://programacionorientadaaobjetosisc.blogspot.mx/constructores-y-estructores.html>

[https://msdn.microsoft.com/es-MX/library/cc433526\(v=vs.71\).aspx](https://msdn.microsoft.com/es-MX/library/cc433526(v=vs.71).aspx)

Redefinición de métodos en clases derivadas.

<http://p-o-o-itsav.blogspot.mx/redefinicion-de-metodos-en-clases.html>

<http://ayudaparaprogramacion.blogspot.mx/2013/11/redefinicion-de-metodos-heredados.html>

UNIDAD IV – Polimorfismo.

4.1 – Definición.

La palabra polimorfismo proviene del griego *poli* (muchos) y *morfos* (formas), y significa que un mismo mensaje enviado a diferentes objetos producirá varias formas diferentes de respuesta. Este es uno de los conceptos esenciales de una programación orientada a objetos. Así como la herencia está relacionada con las clases y su jerarquía, el polimorfismo se relaciona con los métodos.

En programación orientada a objetos, el polimorfismo se refiere a la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos. El único requisito que deben cumplir los objetos que se utilizan de manera polimórfica es saber responder al mensaje que se les envía.

En lenguajes basados en clases y con un sistema de tipos de datos fuerte (independientemente de si la verificación se realiza en tiempo de compilación o de ejecución), es posible que el único modo de poder utilizar objetos de manera polimórfica sea que compartan una raíz común, es decir, una jerarquía de clases, ya que esto proporciona la compatibilidad de tipos de datos necesaria para que sea posible utilizar una misma variable de referencia (que podrá apuntar a objetos de diversas subclases de dicha jerarquía) para enviar el mismo mensaje (o un grupo de mensajes) al grupo de objetos que se tratan de manera polimórfica.

No obstante, algunos lenguajes de programación (Java, C#) permiten que dos objetos de distintas jerarquías de clases respondan a los mismos mensajes, a través de las denominadas interfaces (esta técnica se conoce como composición de objetos). Dos objetos que implementen la misma interfaz podrán ser tratados de forma idéntica, como un mismo tipo de objeto, el tipo definido por la interfaz. Así, distintos objetos podrán intercambiarse en tiempo de ejecución –siempre que sean del mismo tipo–, y además con dependencias mínimas entre ellos. Por estos motivos se considera un buen principio de diseño en programación orientada a objetos el favorecer la composición de objetos frente a la herencia de clases.

En Java las interfaces se declaran mediante la palabra clave `Interface`. Estas se utilizan para lograr la necesaria concordancia de tipos que hace posible el polimorfismo, también como un contrato que debe cumplir cualquier clase que implemente una cierta interfaz, y como una forma de documentación para los desarrolladores. A veces, en la literatura específica sobre Java se habla de "herencia y polimorfismo de interfaces", lo que no concuerda con los conceptos de la programación orientada a objetos porque una clase que implementa una interfaz sólo obtiene su tipo de datos y la obligación de implementar sus métodos, no copia comportamiento ni atributos. Esta terminología puede llevar a confusión, puesto que en Java a menudo se utiliza la mal llamada "herencia de interfaces" para dotar a una clase de uno o varios tipos adicionales, lo que unido a la composición, evite la necesidad de la herencia múltiple y favorezca una utilización más amplia del polimorfismo.

No obstante, el uso de una jerarquía de clases como paso previo, es muy habitual incluso en aquellos lenguajes en los que es posible prescindir de tal jerarquía, ya que, desde una perspectiva conceptual, se puede decir que al pertenecer los "objetos polimórficos" a

subclases de una misma jerarquía, se asegura la equivalencia semántica de los mensajes que se invocarán de modo polimórfico. Por esto, en programación orientada a objetos a veces se denomina al polimorfismo como "polimorfismo de subclase (o de subtipo)".

En resumen, en la programación orientada a objetos, la esencia del polimorfismo no atañe a la clase o prototipo de la que provienen los objetos. Aun así, en los lenguajes basados en clases, es habitual (y en algunos tal vez sea el único modo) que dichos objetos pertenezcan a subclases pertenecientes a una misma jerarquía. Entonces, el polimorfismo debe verse como una forma flexible de usar un grupo de objetos (como si fueran sólo uno). Podría decirse que el polimorfismo en esencia refiere al comportamiento de los objetos, no a su pertenencia a una jerarquía de clases (o a sus tipos de datos).

Además, es importante remarcar que si un cierto grupo de objetos pueden utilizarse de manera polimórfica es porque, en última instancia, todos ellos saben responder a un cierto mensaje (o a varios), pero dado que esos mismos objetos generalmente contendrán otros métodos (que otros objetos en dicho grupo no contienen), difícilmente se pueda decir llanamente que los objetos son polimórficos; lo correcto es decir que esos objetos se pueden utilizar de modo polimórfico para un cierto conjunto de mensajes.

Por ejemplo, podemos crear dos clases distintas: Pez y Ave que heredan de la superclase Animal. La clase Animal tiene el método abstracto mover que se implementa de forma distinta en cada una de las subclases (peces y aves se mueven de forma distinta). Entonces, un tercer objeto puede enviar el mensaje mover a un grupo de objetos Pez y Ave por medio de una variable de referencia de clase Animal, haciendo así un uso polimórfico de dichos objetos respecto del mensaje mover.

El concepto de polimorfismo, desde una perspectiva más general, se puede aplicar tanto a funciones como a tipos de datos. Así nacen los conceptos de funciones polimórficas y tipos polimórficos. Las primeras son aquellas funciones que pueden evaluarse o ser aplicadas a diferentes tipos de datos de forma indistinta; los tipos polimórficos, por su parte, son aquellos tipos de datos que contienen al menos un elemento cuyo tipo no está especificado.

En general, hay tres tipos de polimorfismo:

- [Polimorfismo de sobrecarga](#)
- [Polimorfismo paramétrico](#) (también llamado *polimorfismo de plantillas*)
- [Polimorfismo de inclusión](#) (también llamado *redefinición o subtipado*)

Polimorfismo de sobrecarga

El polimorfismo de sobrecarga ocurre cuando las funciones del mismo nombre existen, con funcionalidad similar, en clases que son completamente independientes una de otra (éstas no tienen que ser clases secundarias de la clase objeto). Por ejemplo, la clase complex, la clase image y la clase link pueden todas tener la función "display". Esto significa que no necesitamos preocuparnos sobre el tipo de objeto con el que estamos trabajando si todo lo que deseamos es verlo en la pantalla.

Por lo tanto, el polimorfismo de sobrecarga nos permite definir operadores cuyos comportamientos varían de acuerdo a los parámetros que se les aplican. Así es posible, por ejemplo, agregar el operador + y hacer que se comporte de manera distinta cuando está haciendo referencia a una operación entre dos números enteros (suma) o bien cuando se encuentra entre dos cadenas de caracteres (concatenación).

Polimorfismo paramétrico

El polimorfismo paramétrico es la capacidad para definir varias funciones utilizando el mismo nombre, pero usando parámetros diferentes (nombre y/o tipo). El polimorfismo *paramétrico* selecciona automáticamente el método correcto a aplicar en función del tipo de datos pasados en el parámetro.

Por lo tanto, podemos por ejemplo, definir varios métodos homónimos de *addition()* efectuando una suma de valores.

- El método *int addition(int,int)* devolvería la suma de dos números enteros.
- *float addition(float, float)* devolvería la suma de dos flotantes.
- *char addition(char, char)* daría por resultado la suma de dos caracteres definidos por el autor.
- etc.

Polimorfismo de subtipado

La habilidad para redefinir un método en clases que se hereda de una clase base se llama **especialización**. Por lo tanto, se puede llamar un método de objeto sin tener que conocer su tipo intrínseco: esto es **polimorfismo de subtipado**. Permite no tomar en cuenta detalles de las clases especializadas de una familia de objetos, enmascarándolos con una interfaz común (siendo esta la clase básica).

Imagine un juego de ajedrez con los objetos *rey*, *reina*, *alfil*, *caballo*, *torre* y *peón*, cada uno heredando el objeto *pieza*.

El método *movimiento* podría, usando polimorfismo de subtipado, hacer el movimiento correspondiente de acuerdo a la clase objeto que se llama. Esto permite al programa realizar el *movimiento.de_pieza* sin tener que verse conectado con cada tipo de pieza en particular.

El polimorfismo suele considerarse el tercer pilar de la programación orientada a objetos, después de la encapsulación y la herencia.

Polimorfismo tiene dos aspectos diferentes:

- En tiempo de ejecución, los objetos de una clase derivada pueden ser tratados como objetos de una clase base en lugares como parámetros de métodos y colecciones o matrices. Cuando ocurre, el tipo declarado del objeto ya no es idéntico a su tipo en tiempo de ejecución.
- Las clases base pueden definir e implementar métodos **virtuales**, y las clases derivadas pueden **invalidarlos**, lo que significa que pueden proporcionar su propia definición e implementación. En tiempo de ejecución, cuando el código de cliente llama al método, CLR busca el tipo en tiempo de ejecución del objeto e invoca esa invalidación del método virtual. Por lo tanto, en el código fuente puede llamar a un método en una clase base y hacer que se ejecute una versión del método de la clase derivada.

Los métodos virtuales permiten trabajar con grupos de objetos relacionados de manera uniforme. Por ejemplo, supongamos que tiene una aplicación de dibujo que permite a un usuario crear varios tipos de formas en una superficie de dibujo. En tiempo de compilación, no sabe qué tipos específicos de formas creará el usuario. Sin embargo, la aplicación tiene que realizar el seguimiento de los distintos tipos de formas que se crean, y tiene que actualizarlos en respuesta a las acciones del mouse del usuario. Para solucionar este problema en dos pasos básicos, puede usar el polimorfismo:

1. Crear una jerarquía de clases en la que cada clase de forma específica deriva de una clase base común.
2. Usar un método virtual para invocar el método apropiado en una clase derivada mediante una sola llamada al método de la clase base.

Ejemplo utilizando **polimorfismo**:

- Primero, cree una clase base llamada **Shape** y clases derivadas como **Rectangle**, **Circle** y **Triangle**.
- Dé a la clase **Shape** un método virtual llamado **Draw** e invalídelo en cada clase derivada para dibujar la forma determinada que la clase representa.
- Cree un objeto **List<Shape>** y agregue Circle, Triangle y Rectangle a él.
- Para actualizar la superficie de dibujo, use un bucle **foreach** para iterar por la lista y llamar al método **Draw** en cada objeto **Shape** de la lista.
- Aunque cada objeto de la lista tenga un tipo declarado de **Shape**, se invocará el tipo en tiempo de ejecución (la versión invalidada del método en cada clase derivada).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        public class Shape //clase base
        {
            // Algunos ejemplos de miembros
            public int X { get; private set; }
            public int Y { get; private set; }
            public int Height { get; set; }
            public int Width { get; set; }

            // Método Virtual
            public virtual void Draw()
            {
                Console.WriteLine("Realizando tareas de dibujo de la clase
base");
            }
        }

        class Circle : Shape //clase derivada
        {
            public override void Draw()
            {
                // Código para dibujar un círculo ...
                Console.WriteLine("\nDibujando un Círculo");
                base.Draw();
            }
        }

        class Rectangle : Shape //clase derivada
        {
            public override void Draw()
            {
                // Código para dibujar un rectángulo ...
                Console.WriteLine("\nDibujando un Rectángulo");
                base.Draw();
            }
        }

        class Triangle : Shape //clase derivada
        {
            public override void Draw()
            {
                // Código para dibujar un triángulo
                Console.WriteLine("\nDibujando un Triángulo");
                base.Draw();
            }
        }
    }
}
```

```

static void Main(string[] args)
{
    // Polimorfismo en el trabajo # 1: un rectángulo,
    // un triángulo y un círculo se pueden usar donde
    // quiera que se espera una Forma.
    // No es necesario cast porque existe una conversión
    // implícita de la clase derivada a su clase base.
    // System.Collections.Generic.List<Shape> shapes =
    // new System.Collections.Generic.List<Shape>();

    List<Shape> shapes = new List<Shape>();
    shapes.Add(new Rectangle());
    shapes.Add(new Triangle());
    shapes.Add(new Circle());

    // Polimorfismo en el trabajo # 2: el método virtual
    // Draw es invocado en cada una de las clases derivadas,
    // no el de la clase base.

    foreach (Shape s in shapes)
    {
        s.Draw();
    }

    // Para mantener la consola abierta en modo de depuración.
    Console.WriteLine("\nPresiona cualquier tecla para Salir.");
    Console.ReadKey();
}

/* SALIDA:
Dibujando un Rectángulo
Realizando tareas de dibujo de la clase base
Dibujando un Triángulo
Realizando tareas de dibujo de la clase base
Dibujando un Círculo
Realizando tareas de dibujo de la clase base
*/
}

```

En C#, cada tipo es polimórfico porque todos los tipos, incluidos los definidos por el usuario, heredan de [Object](#).

4.2 – Clases abstractas.

Estas clases se caracterizan por tener algunos de sus métodos declarados como abstractos, pero ¿qué es un método abstracto?. **Un método abstracto** es aquel que está definido, pero no tiene cuerpo, es decir, le declaramos el nombre, los parámetros y el tipo de retorno, pero no le ponemos llaves "{}".

Las clases abstractas, se utilizan para permitir que otras clases hereden de ella y proporcionar un modelo a seguir.

La declaración de una clase y método abstracto es el siguiente:

```
public abstract class nombreClase
{
    public abstract tipoRetorno nombreMetodo(argumentos (si los lleva));
}
```

Reglas básicas de este tipo de clases:

- Como ya se menciona, una clase se declara abstracta si tiene algún método abstracto.
- Una clase abstracta no puede crear objetos, es decir, no podemos utilizar el operador "new" para una clase abstracta, ya que tiene métodos que no están definidos.
- Si una clase hereda de una clase abstracta, deberá de sobrescribir (implementar) todos los métodos abstractos, si no es así, la clase que hereda deberá ser también abstracta.
- Una clase abstracta puede llevar métodos NO abstractos, pero solo con que exista un método abstracto, la clase deberá ser abstracta.
- Como los métodos estáticos no pueden ser redefinidos, un método abstracto no puede ser estático.
- Una clase abstracta si puede tener constructores. Como sabemos, no es posible crear objetos de una clase abstracta, pero de una clase, la cual herede de una clase abstracta, si se pueden crear objetos, y como sabemos cuando creamos un objeto de una clase que hereda, la primera llamada de su constructor es una llamada al constructor de la clase "padre", lo que permite que aunque no cree un objeto, si se pueda utilizar su constructor.

Este tipo de Clases nos permiten crear “método generales”, que recrean un comportamiento común, pero sin especificar cómo lo hacen.

De estas clases se dice, que su nivel de abstracción es demasiado alto, por lo que su único fin es ser heredadas/extendidas por otras clases.

Ejemplo:

```
1 public abstract class Figura {
2
3     // Atributos:
4     public int numeroLados;
5     public float area;
6     public float volumen;
7
8     // Métodos:
9     abstract public void calcularArea();
10    abstract public void calcularVolumen();
11}
```

En este ejemplo se puede apreciar que toda figura tendrá un método para calcular su Área y Volumen.

Tomando como base la clase abstracta, se pueden implementar clases derivadas que redefinan a los métodos abstractos heredados.

Toda clase derivada debe remplazar los métodos de la clase base abstracta, utilizando la palabra clave **override**, y proveer implementaciones concretas de tales métodos.

Ahora se crea la clase derivada **Esfera**, que “Es una” Figura e implementara los métodos de la clase base abstracta.

```
1 public class Esfera : Figura{
2
3     // Atributos:
4     public float radio;
5     public static float pi = (float)(3.1415);
6
7     // Constructor:
8     public Esfera( float radio ){
9         this.radio = radio;
10        this.numeroLados = 0;
11    }
12
13    // Implementación de Métodos:
14    public override void calcularArea(){
15
16        this.area = (4)*pi*radio*radio; // 4*pi*r²
17        Console.WriteLine("El Área de la Esfera es:" + this.area);
18    }
19
20    public override void calcularVolumen(){
21
22        this.volumen = (4/3)*pi*radio*radio*radio; // (4/3)*pi*radio³
23        Console.WriteLine("El Volumen de la Esfera es:" + this.volumen);
24    }
25}
```

4.3 – Interfaces.

La interfaz es una clase abstracta pura, es decir, una clase donde se indican los métodos, pero no se implementa ninguno. Permite al programador establecer una estructura que ha de seguir toda clase que implemente esta interfaz. En la interfaz solo se indica los nombres de los métodos, sus parámetros y tipos que retornan, pero no el código de cada método.

La definición de una interfaz inicia con la palabra clave `interface` y contiene una lista de métodos y propiedades públicos.

Los métodos que se declaran en una interfaz son públicos de forma implícita, como consecuencia, no se permite el uso del modificador de acceso `public`.

La definición de una interfaz es similar a la definición de una clase abstracta, en el sentido de que sirve como base para una jerarquía y no pueden crearse objetos con ella.

Una interfaz especifica un contrato sintáctico y semántico al que están sujetas todas las clases derivadas. Mas concretamente, en ese contrato la interfaz describe el **qué**, mientras que las clases que implementen la interfaz, describirán el **cómo**.

La principal diferencia entre una *interfaz* y una *clase abstracta* es que una interface proporciona un mecanismo de encapsulación de los protocolos de los métodos sin forzar al usuario a utilizar la herencia.

Mientras que una clase abstracta es mejor para proveer datos y servicios para objetos en una relación jerárquica, una interfaz puede utilizarse para proveer servicios que agrupan objetos distintos que se relacionan solo a través de esa interfaz.

En los casos en que se requiere que una clase derivada herede directamente de varias clases base, debido a que no todos los lenguajes de programación permiten la herencia múltiple, es decir para una clase derivada solo es posible tener una clase base, la alternativa es utilizar interfaces y simular la herencia múltiple.

Para declarar una interfaz se utiliza el siguiente formato:

```
modificador_acceso interface nombreInterfaz {  
    ....  
}
```

Para implementarla en una clase, se utiliza la siguiente forma:

```
modificador_acceso class nombreClase : nombreInterfaz1[, nombreInterfaz2]
```

Una clase puede implementar varias interfaces de los paquetes que se han importado dentro del programa, separando los nombres por comas.

Ejemplo

- Definición de una interfaz:

```
interface Nave {  
    public abstract moverPosicion (int x, int y);  
    public abstract disparar();  
    .....  
}
```

- Uso de la interfaz definida:

```
public class NaveJugador : Nave {  
    public void moverPosicion (int x, int y) {  
        //Implementación del método  
        posActualx = posActualx - x;  
        posActualy = posActualy - y;  
    }  
    public void disparar() {  
        //Implementación del método  
    }  
    ...  
}
```

Las interfaces pueden contener métodos, propiedades, eventos, indizadores o cualquier combinación de estos cuatro tipos de miembros.

Una interfaz no puede contener constantes, campos, operadores, constructores de instancias, destructores o tipos.

Los miembros de interfaz son públicos automáticamente y no pueden incluir modificadores de acceso. Los miembros tampoco pueden ser estáticos.

Para implementar un miembro de interfaz, el miembro correspondiente de la clase de implementación debe ser público, no estático y tener el mismo nombre y firma que el miembro de interfaz.

Cuando una clase o estructura implementa una interfaz, la clase o estructura debe proporcionar una implementación para todos los miembros que define la interfaz.

La propia interfaz no proporciona ninguna funcionalidad que una clase o estructura puedan heredar de la misma forma en que pueden heredar la funcionalidad de la clase base. Sin embargo, si una clase base implementa una interfaz, cualquier clase que se derive de la clase base hereda esta implementación.

Ejemplo sencillo de la definición e implementación de una interfaz en C#:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace InterfaceOperaciones
{
    class Program
    {
        //interfaz
        interface Operaciones
        {
            //métodos
            double Suma();
            double Resta();
            double Multi();
            double Div();
        }

        //campos globales
        public static double Numero1 = 3.6, NUmero2 = 5.9;

        //clase
        class Principal : Operaciones
        {
            //implementacion métodos
            public double Suma()
            { return Numero1 + NUmero2; }

            public double Resta()
            { return Numero1 - NUmero2; }

            public double Multi()
            { return Numero1 * NUmero2; }

            public double Div()
            { return Numero1 / NUmero2; }

            //destructor
            ~Principal()
            { Console.WriteLine("Memoria Objeto Clase Principal
Liberada"); }
        }
    }
}
```

```
static void Main(string[] args)
{
    //creacion de objeto
    Principal p1 = new Principal();

    //ejecucion de métodos y despliegue de resultados
    Console.WriteLine("El resultado de la Suma de los Números es :
{0}", p1.Suma());
    Console.WriteLine("El resultado de la Resta de los Números es :
{0}", p1.Resta());
    Console.WriteLine("El resultado de la Multiplicación de los
Números es : {0}", p1.Multi());
    Console.WriteLine("El resultado de la División de los Números
es : {0}", p1.Div());
}
}
```

4.4 – Variables polimórficas (plantillas).

El término polimórfico (literalmente: muchas formas) se refiere al hecho de que una misma variable puede contener objetos de diferentes tipos (del tipo declarado o de cualquier subtipo del tipo declarado).

El polimorfismo aparece en los lenguajes orientados a objetos en numerosos contextos, las variables polimórficas constituyen justamente un primer ejemplo.

Observemos la manera en que el uso de una variable polimórfica nos ayuda a simplificar nuestro método **imprimir**. El cuerpo de este método es:

```
for (Elemento elemento : elementos)
    elemento.imprimir();
```

En este método recorremos la lista de elementos (contenida en un ArrayList mediante la variable elementos), tomamos cada elemento de la lista y luego invocamos su método imprimir. El uso de herencia en este ejemplo ha eliminado la necesidad de escribir dos ciclos en el método imprimir. La herencia evita la duplicación de código no sólo en las clases servidoras sino también en las clases clientes de aquellas.

Un ArrayList declarado puede contener objetos de cualquier tipo. De forma general se crea de la siguiente manera:

```
ArrayList nombreArray = new ArrayList();
```

Ejemplo del uso de variables polimórficas:

```
public class CD
{
    private String title;
    private String artist;
    private String comment;
    CD(String theTitle, String theArtist)
    {
        title = theTitle;
        artist = theArtist;
        comment = " ";
    }
    void setComment(String newComment)
    { ... }
    String getComment()
    { ... }
    void print()
    { ... }
    ...
}
```

```
public class DVD
{
    private String title;
    private String director;
    private String comment;
    DVD(String theTitle, String theDirector)
    {
        title = theTitle;
        director = theDirector;
        comment = " ";
    }
    void setComment(String newComment)
    { ... }
    String getComment()
    { ... }
    void print()
    { ... }
    ...
}

class Database {
    private ArrayList<CD> cds;
    private ArrayList<DVD> dvds;
    ...
    public void list()
    {
        for(CD cd : cds) {
            cd.print();
            System.out.println(); }
        for(DVD dvd : dvds) {
            dvd.print();
            System.out.println(); }
    }
}
```

4.5 – Reutilización de código.

La **reutilización de código** se refiere al comportamiento y a las técnicas que garantizan que una parte o la totalidad de un **programa informático** existente se pueda emplear en la construcción de otro programa. De esta forma se aprovecha el trabajo anterior, se economiza tiempo, y se reduce la redundancia.

La manera más fácil de reutilizar código es copiarlo total o parcialmente desde el programa antiguo al programa en desarrollo. Pero es trabajoso mantener múltiples copias del mismo código, por lo que en general se elimina la redundancia dejando el código reusable en un único lugar, y llamándolo desde los diferentes programas.

Este proceso se conoce como **abstracción**. La abstracción puede verse claramente en las **bibliotecas de software**, en las que se agrupan varias operaciones comunes a cierto dominio para facilitar el desarrollo de programas nuevos. Hay bibliotecas para convertir información entre diferentes formatos conocidos, acceder a dispositivos de almacenamiento externos, proporcionar una interfaz con otros programas, manipular información de manera conocida (como números, fechas, o cadenas de texto).

Para que el código existente se pueda reutilizar, debe definir alguna forma de comunicación o interfaz. Esto se puede dar por llamadas a una subrutina, a un objeto, o a una clase.

Para reutilizar el código creamos nuevas clases pero, en lugar de partir de cero, partimos de clases, relacionadas con nuestra clase, que han sido ya creadas y depuradas.

Una forma de hacer esto es crear objetos de nuestras clases existentes dentro de la nueva clase. Esto se conoce como composición porque la nueva clase está compuesta de objetos de clases existentes. Estamos reutilizando la funcionalidad del código, y no la forma.

Otra forma es crear una nueva clase como un tipo de una clase ya existente. Tomamos la forma de la clase existente y añadimos código a la nueva, sin modificar la clase existente. Esta forma de crear nuevos objetos se llamada herencia, y lo que hacemos es extender la clase en la que nos basamos para crear la nueva.

Composición:

Hasta ahora hemos usado la composición de cierta manera, por ejemplo, cuando hacemos una interfaz gráfica de usuario, nuestra clase de interfaz gráfica esta compuesta por un frame, unos panel, unos botones, etc. Todos estos objetos componen el objeto de interfaz gráfica, es decir que la composición consiste en poner manejadores de objetos dentro de nuestra clase, estos manejadores de objetos no serán otra cosa que instancias de las clases en las que nos estamos basando para crear la nueva clase.

Recordemos que la forma para determinar cuando usar composición es cuando podemos decir que nuestra nueva clase “tiene un” elemento de otro tipo de objetos, por ejemplo un cronómetro tiene: horas, minutos y segundos, es decir que una clase Cronometro esta compuesta por otras clases llamadas: Horas, Minutos y Segundos.

Veamos como seria esta clase:

```
public class Cronometro {
    Horas h;
    Minutos m;
    Segundos s;

    String cadena;
    int seg,min,hor;

    public Cronometro() {
        seg=0;
        min=0;
        hor=0;
        h = new Horas();
        m = new Minutos();
        s = new Segundos();
        cadena = new String("0 : 0 : 0");
    }

    public String avanzar(){
        seg = s.forward();
        if(seg==0){
            min=m.forward();
            if(min==0){
                hor=h.forward();
            }
        }
        cadena = hor + " : " + min + " : " + seg;
        return cadena;
    }

    public String reset(){
        seg = s.reset();
        min = m.reset();
        hor = h.reset();
        cadena = hor + " : " + min + " : " + seg;
        return cadena;
    }
}
```

Nuestra clase Cronometro está compuesta entre otras cosas por objetos del tipo Horas, Minutos y Segundos, y a través del constructor de nuestra clase creamos las instancias de cada una de ellas.

Herencia:

En los lenguajes de programación orientados a objetos, aunque no establezcamos de manera explícita la herencia siempre está presente, ya que todas las clases que creamos heredan de la clase Object, por eso es valido decir que todo es un objeto. La sintaxis para la composición es obvia pero, para realizar la herencia, hay una forma claramente distinta.

Cuando heredamos, estamos diciendo "Esta nueva clase es como esa clase antigua", por ejemplo, al decir que la clase Horas "es una" UnidadDeTiempo. Afirmamos esto en el código cuando la creamos, y cuando hagamos esto, obtendremos automáticamente todos los datos miembros y métodos de la clase base.

Primero veamos como seria la clase UnidadDeTiempo:

```
public class UnidadDeTiempo {
    int valor;
    int tope;

    public int forward(){
        if(valor == tope)
            valor=0;
        else
            valor++;
        return valor;
    }

    public int reset(){
        valor=0;
        return valor;
    }
}
```

Y nuestra clase Horas:

```
public class Horas : UnidadDeTiempo{
    public Horas() {
        this.valor=0;
        this.tope=23;
    }
}
```

De esta manera sin necesidad de tener que escribir nuevamente todo el código de UnidadDeTiempo lo tenemos disponible en la clase Horas, pero que partes tenemos disponibles?, todos los atributos y los métodos de la clase padre están disponibles en la clase hija?

Dependiendo de los modificadores de acceso o visibilidad de estos, se determinara que estará disponible para las clases hijas que heredan de una clase padre.

El código de nuestra clases hijas no tienen porque limitarse solo al código heredado, de hecho casi siempre la herencia se hace para extender la funcionalidad de las clases heredadas añadiendo nuevos métodos y atributos.

La composición y la herencia:

Tanto la composición como la herencia permiten poner sub-objetos dentro de tu nueva clase. Podríamos preguntarnos ¿cuál es la diferencia entre las dos?, y ¿cuándo elegir una en lugar de la otra?.

La composición es generalmente usada cuando deseamos las características de una clase existente dentro de una nueva clase, pero no su interfaz. Es decir, ponemos una para poder usarla para implementar características de nuestra nueva clase, pero el

usuario de esa nueva clase verá la interfaz que hemos definido en lugar de la interfaz del objeto insertado. Los objetos miembros usan la implementación ocultándose a sí mismos.

Cuando heredamos, estamos tomando una clase existente y creando una versión especial de esa clase. En general, esto significa que estamos tomando una clase de propósito general, especializándola para un caso o necesidad particular.

La relación es-un viene expresada por la herencia, y la relación tiene-un viene expresada por la composición.

REFERENCIAS BIBLIOGRAFICAS UNIDAD 4

Definición:

[https://es.wikipedia.org/wiki/Polimorfismo_\(inform%C3%A1tica\)](https://es.wikipedia.org/wiki/Polimorfismo_(inform%C3%A1tica))

<http://es.ccm.net/contents/413-oop-polimorfismo>

<https://msdn.microsoft.com/es-es/library/ms173152.aspx>

Clase Abstracta:

<https://emartini.wordpress.com/.../poo-clases-abstractas-interfaces-y-herencia-multiple/>

<http://picarcodigo.blogspot.com/2012/10/clases-abstractas.html>

[https://msdn.microsoft.com/es-es/library/aa645615\(v=vs.71\).aspx](https://msdn.microsoft.com/es-es/library/aa645615(v=vs.71).aspx)

Interfaces:

<http://www.androidcurso.com/index.php/tutoriales-java-esencial/686-clases-abstractas-interfaces-y-herencia-multiple>

[https://es.wikipedia.org/wiki/Interfaz_\(Java\)](https://es.wikipedia.org/wiki/Interfaz_(Java))

<https://docs.microsoft.com/es-es/dotnet/articles/csharp/programming-guide/interfaces/>

Variables polimórficas:

<http://fjglez90.blogspot.mx/2012/06/44-variables-polimorficas-plantillas.html>

<http://consejostallerdeprogramacion.blogspot.mx/2015/10/variables-polimorficas-en-javas.html>

Reutilización de código:

https://es.wikipedia.org/wiki/Reutilizaci%C3%B3n_de_c%C3%B3digo

<http://apdaza-poo.blogspot.mx/2008/04/reutilizaci%C3%B3n-del-c%C3%B3digo-y-relaciones.html>

<http://itzoisc.blogspot.mx/2012/06/33-reutilizaci%C3%B3n-de-miembros-heredados.html>

UNIDAD V – Excepciones.

5.1 – Definición.

La ejecución de cualquier programa puede enfrentar errores imprevistos. Los problemas podrían no ser el resultado de una anomalía en el código. En lugar de eso, por ejemplo, un programa pudiera necesitar acceder a un dispositivo aun no instalado en la computadora, o un disco puede estar lleno y es imposible crear un archivo, o se trata de abrir un archivo inexistente en una ruta de acceso especificada.

El método tradicional para la detección de errores es hacer que el programa devuelva un código de error cuando falla una operación. Sin embargo, el problema de este enfoque es que puede ocurrir un error no previsto o que se sale del ámbito del programa, en cuyo caso el programa se bloqueará de forma abrupta.

A una situación anómala o condición de error que ocurre durante el tiempo de ejecución de un programa se le denomina **excepción**.

El nombre "excepción" viene del hecho de que aunque un problema puede ocurrir lo hace con poca frecuencia, esto es, de manera excepcional.

5.2 – Tipos de excepciones.

El mecanismo de manejo de excepciones sólo permite lanzar y atrapar objetos de la clase base [Exception](#) (del espacio de nombres [System](#)) y sus clases derivadas.

La clase [Exception](#) es la clase base de la jerarquía de excepciones del Framework del .NET.

Dos de las más importantes clases derivadas de [Exception](#) son:

1. [SystemException](#)
2. [ApplicationException](#)

El Common Language Runtime (CLR) puede generar excepciones en cualquier punto durante la ejecución del programa. Tales excepciones son llamadas [excepciones en tiempo de ejecución](#) y derivan de la clase [SystemException](#).

Por ejemplo, si un programa intenta acceder a un subíndice de arreglo fuera de rango, el CLR lanza una excepción de tipo [OutOfRangeException](#) (una clase derivada de [SystemException](#)).

De manera similar, ocurre una excepción en tiempo de ejecución cuando un programa utiliza una referencia a objeto para manipular un objeto que aun no existe (esto es, que la referencia tiene un valor [null](#)). El intento de utilizar una referencia [null](#) causa una [NullReferenceException](#).

En una jerarquía de excepciones relacionadas, un manejador [catch](#) puede atrapar excepciones de un tipo particular o utilizar la clase base para atrapar excepciones derivadas.

Por ejemplo, un manejador [catch](#) que especifica un parámetro de excepción de tipo [Exception](#) también puede atrapar excepciones de todas las clases derivadas de [Exception](#). Esto permite el procesamiento polimórfico de excepciones relacionadas en una jerarquía.

5.3 – Propagación de excepciones.

El Framework .NET y el lenguaje C# están orientados a componentes, de modo que el mecanismo de excepciones fue deliberadamente diseñado para asegurarse de que una excepción lanzada mientras el programa ejecuta código en un componente no es pasada por alto o perdida por un programa cliente.

Si una excepción es lanzada mientras se ejecuta el código de un componente, el componente puede manejar la excepción, en cuyo caso el cliente puede no ser advertido de que se lanzó la excepción. Sin embargo, si el componente no maneja la excepción, se pasa de nuevo al programa cliente.

Es posible que un manejador de excepciones, al recibir una excepción, decida que no puede procesar esa excepción o que puede procesarla sólo parcialmente. En ese caso, el manejador de excepciones puede diferir el manejo de la excepción lanzándola hacia otro manejador de excepciones.

Las características de control de excepciones del lenguaje C# le ayudan a afrontar cualquier situación inesperada o excepcional que se produce cuando se ejecuta un programa.

El control de excepciones usa las palabras clave `try`, `catch` y `finally` para intentar realizar acciones que pueden no completarse correctamente, para controlar errores cuando decide que es razonable hacerlo y para limpiar recursos más adelante.

Las excepciones las puede generar Common Language Runtime (CLR), .NET Framework, cualquier biblioteca de terceros o el código de aplicación. Las excepciones en el código se crean mediante el uso de la palabra clave `throw`.

En muchos casos, una excepción la puede no producir un método al que el código ha llamado directamente, sino otro método más bajo en la pila de llamadas. Cuando esto sucede, CLR desenredar la pila, busca un método con un bloque `catch` para el tipo de excepción específico y ejecuta el primer bloque `catch` que encuentra. Si no encuentra ningún bloque `catch` adecuado en cualquier parte de la pila de llamadas, finalizará el proceso y mostrará un mensaje al usuario.

5.4 – Gestión de excepciones.

La gestión de excepciones se lleva a cabo en varias fases:

- Cuando se detecta un error, se lanza ([throw](#)) una excepción.
- Cuando esto ocurre, existe la opción de atrapar ([catch](#)) la excepción y manejar el error.

5.4.1.- Manejo de excepciones.

El manejo de excepciones se implementa por medio de la palabra clave [catch](#).

En un programa que maneje excepciones no se utiliza [catch](#) de manera aislada, sino que siempre está asociada a la palabra [try](#).

[try...catch...finally](#)

La instrucción [try...catch...finally](#) consiste de un bloque [try](#) (intentar) seguido por uno o más cláusulas [catch](#), las cuales especifican manejadores para diferentes excepciones.

La cláusula [catch](#) puede utilizarse sin argumentos, en cuyo caso atrapa cualquier tipo de excepción.

También puede tomar un argumento objeto derivado de [System.Exception](#), en cuyo caso maneja una excepción específica.

La clase [System.Exception](#) tiene cuatro diferentes constructores:

1. [public Exception\(\) ;](#)
2. [public Exception\(String\) ;](#)
3. [protected Exception\(SerializationInfo, StreamingContext\) ;](#)
4. [public Exception\(String, Exception\) ;](#)

Un uso común es obtener y utilizar recursos en un bloque [try](#), manejar circunstancias excepcionales en un bloque [catch](#), y liberar los recursos en el bloque [finally](#).

El bloque [finally](#) se ejecuta independientemente de que el correspondiente bloque [try](#) se ejecute exitosamente o no.

Esto hace del bloque [finally](#) un lugar ideal para colocar código que libera recursos adquiridos y manipulados en el correspondiente bloque [try](#).

Si el bloque [try](#) se ejecuta exitosamente, el bloque [finally](#) se ejecuta inmediatamente después de que termina el bloque [try](#).

Si ocurre una excepción en el bloque [try](#), el bloque [finally](#) se ejecuta inmediatamente después del bloque [catch](#) que completa el manejo de la excepción.

Propiedades de las Excepciones.

La clase [Exception](#) y sus clases derivadas tienen varias propiedades, entre otras:

1. [Data](#)
2. [HelpLink](#)
3. [HResult](#)
4. [InnerException](#)
5. [IsTransient](#)
6. [Message](#)
7. [Source](#)
8. [StackTrace](#)
9. [TargetSite](#)

Por ejemplo, la propiedad [Message](#) almacena los mensajes de error asociados con un objeto de la clase [Exception](#). Este mensaje puede ser un mensaje predefinido asociado con el tipo de excepción o un mensaje personalizado pasado a un constructor del objeto excepción cuando el objeto es construido.

La propiedad [StackTrace](#) contiene una cadena que representa la pila de llamadas a métodos. El ambiente de tiempo de ejecución mantiene una lista de llamadas a métodos que han sido hechas hasta un momento dado.

La cadena de [StackTrace](#) representa esta lista secuencial de métodos que no han finalizado su procesamiento al momento de ocurrir la excepción. La ubicación exacta en la cual ocurre la excepción en el programa se denomina [punto de lanzamiento de la excepción](#).

Generalmente, los programadores de bibliotecas de clases utilizan la propiedad [InnerException](#) para "envolver" objetos de excepción atrapados en su código y luego lanzar nuevos tipos de excepción que son específicos de sus bibliotecas.

Y así sucesivamente, todas las propiedades proporcionan información útil para el programador.

Programa ejemplo de la excepción: [IndexOutOfRangeException](#)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ExcepcionIndexOutOfRange
{
    class Program
    {
        static void Main(string[] args)
        {
            //variables auxiliares
            int[] arreglo = new int[10];

            Random R = new Random();

            try
            {
                //generación aleatoria del vector
                for (int x = 0; x < arreglo.Length; x++)
                {
                    arreglo[x] = R.Next(10, 99);
                }

                //ordenamiento del vector
                Array.Sort(arreglo);

                //despliegue del vector ordenado
                Console.WriteLine("\n\nARREGLO ORDENADO");
                for (int x = 0; x < arreglo.Length; x++)
                {
                    Console.WriteLine("\t{0}.-{1}", x+1, arreglo[x]);
                }
                Console.Write("\npresione <enter> para continuar...");
                Console.ReadKey();

                arreglo[15] = 12345; //acceso indice fuera de rango
            }
            catch (IndexOutOfRangeException i)
            {
                Console.WriteLine("\n\nMensaje del Error: " + i.Message);
                Console.WriteLine("\nRuta del Error: " + i.StackTrace);
                Console.Write("\npresione <enter> para continuar...");
                Console.ReadKey();
            }
            finally
            {
                Console.WriteLine("\n\nEl bloque finally siempre se ejecuta.");
                Console.Write("\nPresione <enter> para Salir del Programa...");
                Console.ReadKey();
            }
        }
    }
}
```

Programa ejemplo de la excepción: [FormatException](#)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ExcepcionFormato
{
    class Program
    {
        static void Main(string[] args)
        {
            //variables auxiliares
            int B, P;
            double R;

            try
            {
                //captura de datos
                Console.Write("\nAlimenta el valor de la base: ");
                B = Int32.Parse(Console.ReadLine());
                Console.Write("\nAlimenta el valor de la potencia: ");
                P = Int32.Parse(Console.ReadLine());

                //operacion
                R = Math.Pow(B,P);

                //despliegue de resultado
                Console.WriteLine("\nEl resultado es: {0}", R);
                Console.Write("\npresione <enter> para continuar...");
                Console.ReadKey();
            }

            catch (FormatException f)
            {
                Console.WriteLine("\nMensaje del Error: " + f.Message);
                Console.WriteLine("\nRuta del Error: " + f.StackTrace);
                Console.Write("\npresione <enter> para continuar...");
                Console.ReadKey();
            }

            finally
            {
                Console.WriteLine("\n\nEl bloque finally siempre se ejecuta.");
                Console.Write("\nPresione <enter> para Salir del Programa...");
                Console.ReadKey();
            }
        }
    }
}
```

Programa ejemplo de la excepción: `OverflowException`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ExcepcionOverflow
{
    class Program
    {
        static void Main(string[] args)
        {
            //variables auxiliares
            byte num1, num2, suma;

            try
            {
                //captura de datos
                Console.WriteLine("\nAlimenta el valor del 1er número: ");
                num1 = Byte.Parse(Console.ReadLine());
                Console.WriteLine("\nAlimenta el valor del 2do número: ");
                num2 = Byte.Parse(Console.ReadLine());

                //operacion
                suma = (byte)(num1 + num2);

                //despliegue de resultado
                Console.WriteLine("\nEl resultado de la suma de los números tipo byte
es: {0}", suma);
                Console.WriteLine("\npresione <enter> para continuar...");
                Console.ReadKey();
            }

            catch (OverflowException s)
            {
                Console.WriteLine("\nMensaje del Error: " + s.Message);
                Console.WriteLine("\nRuta del Error: " + s.StackTrace);
                Console.WriteLine("\npresione <enter> para continuar...");
                Console.ReadKey();
            }

            finally
            {
                Console.WriteLine("\n\nEl bloque finally siempre se ejecuta.");
                Console.WriteLine("\nPresione <enter> para Salir del Programa...");
                Console.ReadKey();
            }
        }
    }
}
```

5.4.2.- Lanzamiento de excepciones.

Independientemente de que la mayoría de las excepciones del Framework del .NET son lanzadas automáticamente por el CLR, el lanzamiento de excepciones también se puede realizar en un programa, por medio de la instrucción `throw`.

Programa ejemplo de la excepción: `DivideByZeroException`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ExcepcionDivideByZero
{
    class Program
    {
        static double Divide(double x, double y)
        {
            if (y == 0)
                throw new DivideByZeroException();
            else return x / y;
        }
        static void Main(string[] args)
        {
            //variables auxiliares
            double a, b, resultado;

            try
            {
                //captura de datos
                Console.WriteLine("\nAlimenta el valor del 1er número: ");
                a=Double.Parse(Console.ReadLine());
                Console.WriteLine("\nAlimenta el valor del 2do número: ");
                b=Double.Parse(Console.ReadLine());

                //llamada al metodo
                resultado = Divide(a,b);
                //despliegue de resultado
                Console.WriteLine("\nEl número {0} dividido por {1} da como resultado:
{2}", a,b,resultado);
                Console.WriteLine("\npresione <enter> para continuar...");
                Console.ReadKey();
            }
            catch(DivideByZeroException d)
            {
                Console.WriteLine("\nMensaje del Error: " + d.Message);
                Console.WriteLine("\nRuta del Error: " + d.StackTrace);
                Console.WriteLine("\npresione <enter> para continuar...");
                Console.ReadKey();
            }
            finally
            {
                Console.WriteLine("\n\nEl bloque finally siempre se ejecuta.");
                Console.WriteLine("\nPresione <enter> para Salir del Programa...");
                Console.ReadKey();
            }
        }
    }
}
```

```

    }
}
}

```

Programa ejemplo de la excepción: [ArgumentOutOfRangeException](#)

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ExcepcionArgumentoFueraRango
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                int Base;
                double Resultado;
                Console.WriteLine("Alimenta un numero entero: ");
                Base = Int16.Parse(Console.ReadLine());
                if (Base < 100)
                {
                    Resultado = Math.Pow(Base, 3);
                    Console.WriteLine("\nEl resultado de evaluar {0} al cubo es: {1}",
Base, Resultado);
                    Console.ReadLine();
                }
                else throw new ArgumentOutOfRangeException();
            }
            catch (ArgumentOutOfRangeException ax)
            {
                Console.WriteLine("\nMensaje del Error: "+ ax.Message);
                Console.WriteLine("\nRuta del Error: "+ ax.StackTrace);
                Console.ReadLine();
            }
            catch (FormatException fx)
            {
                Console.WriteLine("\nMensaje del Error: "+ fx.Message);
                Console.WriteLine("\nRuta del Error: "+ fx.StackTrace);
                Console.ReadLine();
            }
            finally
            {
                Console.WriteLine("\nPresione ENTER para Salir del Programa");
                Console.ReadLine();
            }
        }
    }
}

```

5.5 – Creación y manejo de excepciones definidas por el usuario.

En algunos casos puede ser necesario crear tipos de excepción específicas para los errores que ocurren en los programas, por lo que deben crearse [clases de excepción definidas por el programador](#).

Las clases de excepción definidas por el programador deben derivar directa o indirectamente de la clase [ApplicationException](#) del espacio de nombres [System](#).

Antes de crear una clase de excepción definida por el programador, es recomendable verificar la existencia de dicha clase en el .NET Framework, y solamente crear clases de excepción para manejar las nuevas excepciones de manera diferente a otros tipos de excepción existentes.

Programa ejemplo de la excepción: [NegativosException](#)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ExcepcionSenoAngulo
{
    class NegativosException:ApplicationException
    {
        //constructor
        public NegativosException()
            : base("NO se puede calcular el seno de un Angulo Negativo!!!")
        { }

        static void Main(string[] args)
        {
            int Angulo;
            double Resultado;

            try
            {
                Console.Write("Alimenta un numero entero: ");
                Angulo = Int16.Parse(Console.ReadLine());
                if (Angulo < 0)
                {
                    throw new NegativosException();
                }
                else
                {
                    Resultado = Math.Sin(Angulo);
                    Console.WriteLine("El Seno del Angulo {0} es : {1}", Angulo,
Resultado);
                    Console.WriteLine("\npresione ENTER para continuar");
                    Console.ReadLine();
                }
            }
        }
    }
}
```

```
        catch (NegativosException n)
        {
            Console.WriteLine("\nMensaje del error: " + n.Message);
            Console.WriteLine("\nRuta del error: " + n.StackTrace);
            Console.WriteLine("\npresione ENTER para continuar");
            Console.ReadLine();
        }
        catch (FormatException f)
        {
            Console.WriteLine("\nMensaje del error: " + f.Message);
            Console.WriteLine("\nRuta del error: " + f.StackTrace);
            Console.WriteLine("\npresione ENTER para continuar");
            Console.ReadLine();
        }
        finally
        {
            Console.WriteLine("\n\nPresione ENTER para Salir del Programa.");
            Console.ReadLine();
        }
    }
}
```

Programa ejemplo de la excepción: [ExcepcionEcuacionCuadratica](#)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        class ExcepcionEcuacionCuadratica : ApplicationException
        {
            //constructor
            public ExcepcionEcuacionCuadratica()
                : base("La Ecuación Cuadratica NO tiene Solución!!!")
            { }
        }

        static void Main(string[] args)
        {
            //variables auxiliares
            int a, b, c, x1, x2, opcion=0, discriminante;

            do
            {
                try
                {
                    Console.Clear();
                    Console.WriteLine("\t\tM E N U");
                    Console.WriteLine("\t1.-Calcular Ecuación Cuadratica.");
                    Console.WriteLine("\t2.-Salir del Programa.");
                    Console.Write("\n\tAlimenta tu opción: ");
                    opcion = Int16.Parse(Console.ReadLine());

                    switch (opcion)
                    {
                        case 1:
                            //captura de datos
                            Console.Write("\nAlimenta el valor de A: ");
                            a = Int16.Parse(Console.ReadLine());
                            Console.Write("\nAlimenta el valor de B: ");
                            b = Int16.Parse(Console.ReadLine());
                            Console.Write("\nAlimenta el valor de C: ");
                            c = Int16.Parse(Console.ReadLine());

                            discriminante = (int)(Math.Pow(b, 2) - 4 * a * c);

                            if (discriminante < 0)
                                throw new ExcepcionEcuacionCuadratica();
                            else
                            {
                                //calculo de resultados
                                x1 = (int)(-b + Math.Sqrt(Math.Pow(b, 2) - 4 * a *
c)) / 2 * a;
                                x2 = (int)(-b - Math.Sqrt(Math.Pow(b, 2) - 4 * a *
c)) / 2 * a;

```

```

        //despliegue de resultados
        Console.WriteLine("\n\nEl valor de X1 es: {0}", x1);
        Console.WriteLine("El valor de X2 es: {0}", x2);
        Console.Write("\npresione <enter> para
Continuar...");

        Console.ReadKey();
    }
    break;

    case 2:
        Console.Write("\npresione <enter> para Salir del
Programa...");

        Console.ReadKey();

        break;

    default:
        Console.Write("\nOpción Inexistente. Presione <enter> para
Continuar...");

        Console.ReadKey();
        break;
    }
}

catch (ExcepcionEcuacionCuadratica ec)
{
    Console.WriteLine("\nMensaje del Error: " + ec.Message);
    Console.WriteLine("\nRuta del Error: " + ec.StackTrace);
    Console.Write("\npresione <enter> para Continuar...");
    Console.ReadKey();
}
catch (FormatException f)
{
    Console.WriteLine("\nMensaje del Error: " + f.Message);
    Console.WriteLine("\nRuta del Error: " + f.StackTrace);
    Console.Write("\npresione <enter> para Continuar...");
    Console.ReadKey();
}

finally
{
    Console.WriteLine("\n\nEl bloque del finally siempre se ejecuta.");
    Console.Write("\npresione <enter> para Regresar al Menu...");
    Console.ReadKey();

}

}while(opcion!=2);
}
}
}

```

REFERENCIAS BIBLIOGRAFICAS UNIDAD 5

Definición:

https://es.wikipedia.org/wiki/Manejo_de_excepciones
p-o-o-itsav.blogspot.com/2012/02/51-definicion-de-excepciones.html

Tipos de excepciones:

www.itnuevolaredo.edu.mx/takeyas/apuntes/poo/Apuntes/05.-%20Excepciones.pdf
<http://itzoisc.blogspot.mx/2012/06/52-tipos-de-excepciones.html>

Propagación de excepciones:

<http://vcalpena.wordpress.com/4-excepciones/>
http://www.unirioja.es/cu/jearansa/0910/archivos/EIPR_Tema05.pdf

Gestión de excepciones:

https://universidad-de-los-andes.gitbooks.io/fundamentos-de-programacion/content/Nivel4/5_ManejoDeLasExcepciones.html
<https://www.fdi.ucm.es/profesor/gmendez/docs/prog0607/Tema5-Excepciones.pdf>

Creación y manejo de excepciones definidas por el usuario:

[https://msdn.microsoft.com/es-es/library/87cdya3t\(v=vs.110\).aspx](https://msdn.microsoft.com/es-es/library/87cdya3t(v=vs.110).aspx)
<http://xtremepoo.blogspot.mx/2009/06/excepciones-definidas-por-el-usuario.html>

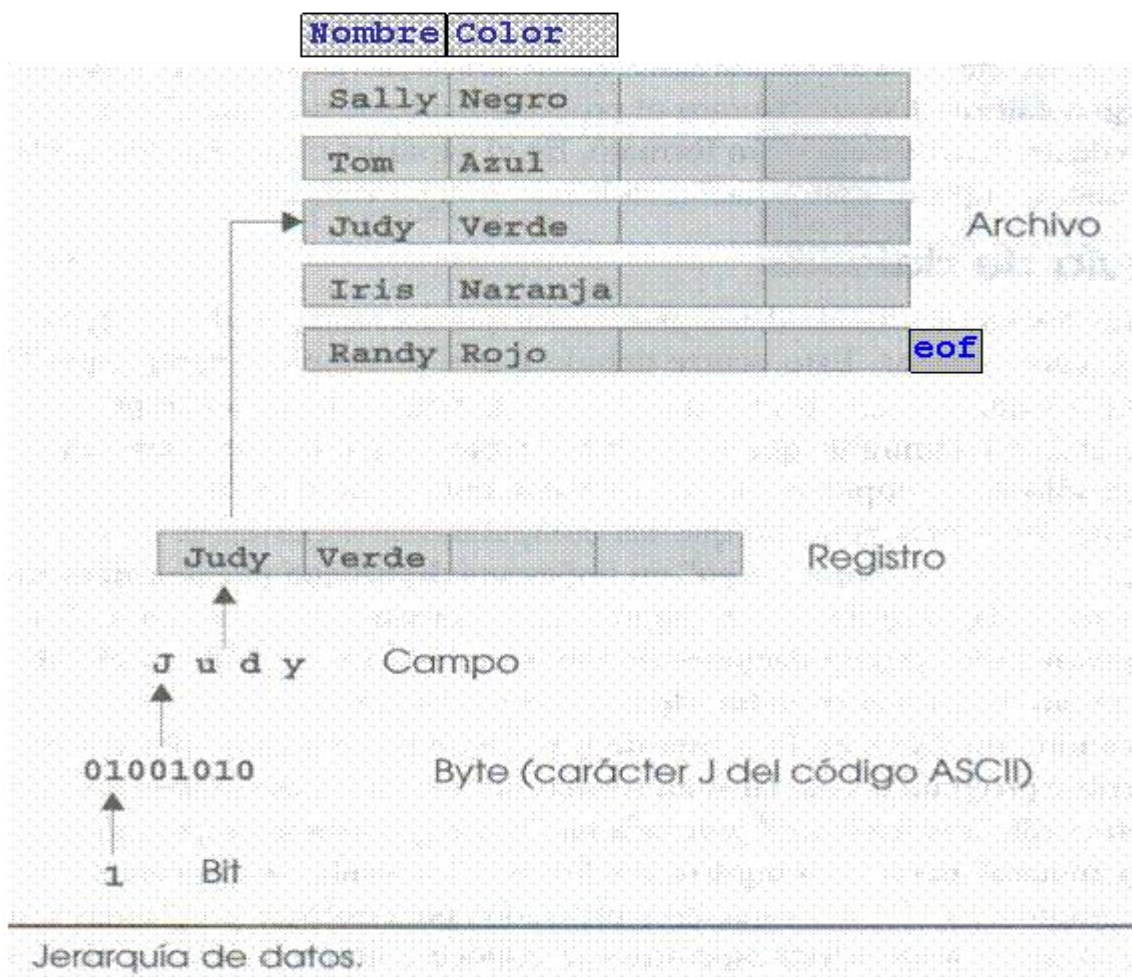
UNIDAD VI – Flujos y Archivos.

6.1 – Definición.

Un archivo es una colección de datos guardados en un dispositivo de almacenamiento permanente.

Aunque C# ve a los archivos como un flujo de bytes, es conveniente concebirlos como un conjunto de registros que poseen una marca de fin de archivo (eof).

La información de un archivo se organiza en registros, los registros en campos, los campos en bytes, y los bytes en bits.



FLUJOS

Para que un programa pueda manejar un archivo en un dispositivo de almacenamiento permanente, como por ejemplo un disco, primero debe crearse un flujo.

Un flujo es un conducto a través del cual se transportarán los datos hacia o desde el dispositivo de almacenamiento. Los datos fluirán entre la memoria RAM de la computadora y el dispositivo de almacenamiento.

Flujo de Entrada:

Si los datos van a enviarse desde el dispositivo de almacenamiento hacia la memoria de la computadora, el flujo es de ENTRADA (Input, en inglés).

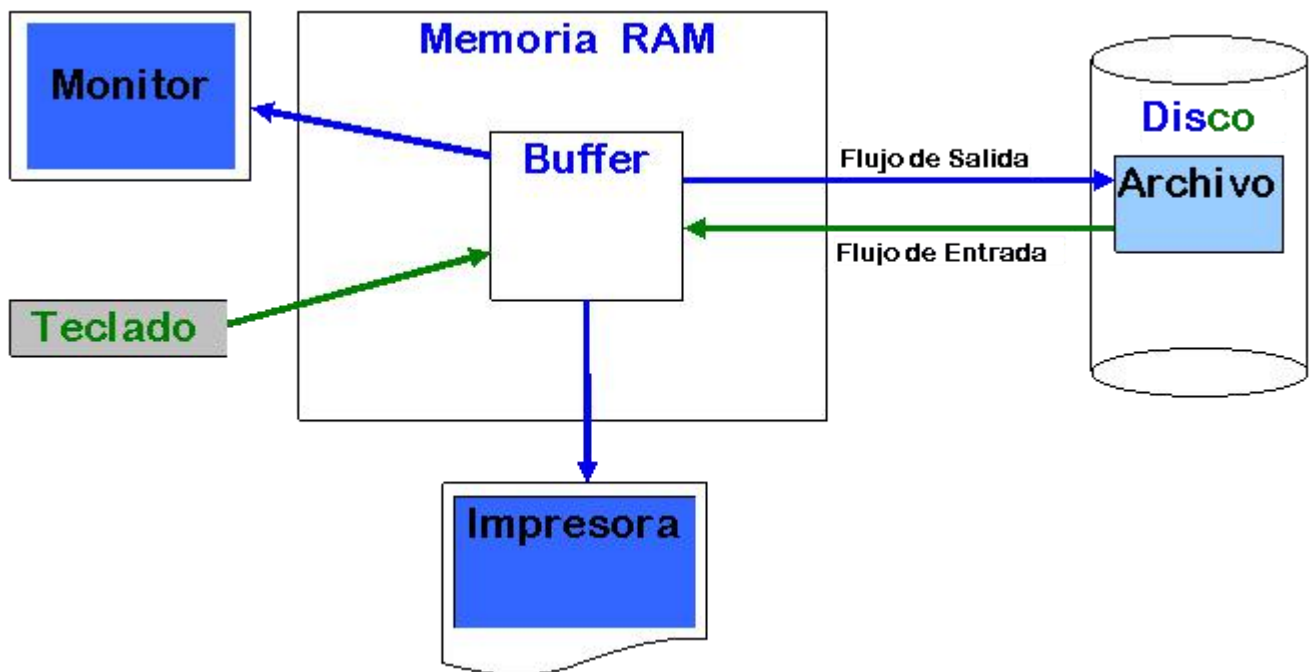
A las operaciones de ENTRADA se les conoce como de LECTURA (Read, en Inglés).

Flujo de Salida:

Si los datos van a enviarse desde la memoria de la computadora hacia el dispositivo de almacenamiento, se trata de un flujo de SALIDA (Output, en inglés).

A las operaciones de SALIDA se les conoce como de ESCRITURA (Write, en Inglés).

En la siguiente figura se esquematiza la relación existente entre los dispositivos de Entrada/Salida y los flujos.



6.2 – Clasificación: Archivos de texto y binarios.

Dependiendo del tipo de datos que manejan en sus registros, los archivos se clasifican en **archivos de texto y archivos binarios**.

Archivos de texto.

Los datos en los archivos de texto se graban como secuencias de bytes.

Por ejemplo, el dato 123456 se graba como una secuencia de 6 bytes y no como un entero, por lo que no pueden realizarse operaciones matemáticas con este tipo de datos.

El manejo de archivos de texto se puede llevar a cabo por medio de dos tipos de flujos: de bytes y de caracteres.

Archivos binarios.

Cuando se requiere efectuar operaciones con datos de alguno de los tipos primitivos del lenguaje (bool, byte, double, float, int, long, short, etc.), tales datos deberán escribirse y leerse en formato binario.

6.3 – Operaciones básicas y tipos de acceso.

Las operaciones básicas en archivos son:

- Creación
- Apertura
- Lectura
- Escritura
- Recorrido
- Cierre

a)Archivos de texto.

El manejo de archivos de texto se puede llevar a cabo por medio de dos tipos de flujos: de **bytes** y de **caracteres**.

1.-Archivos de Texto con Flujos de Bytes.

Para escribir o leer datos de tipo byte en un archivo se declara un flujo de la clase **FileStream** , cuyos constructores son:

FileStream (string Nombre , **FileMode** Modo)
FileStream (string Nombre , **FileMode** Modo , **FileAccess** Acceso)

Donde:

Nombre es el nombre del archivo en disco, incluyendo la trayectoria.

Ejemplo:

"C: \\ POOISC \\ ARCHIVOS \\ archivo.txt"

Modo es un valor del tipo enumerado **FileMode**; puede tomar uno de los siguientes valores:

Valor	Acción
CreateNew	Crea un nuevo archivo. Si el archivo existe, lanzará una excepción del tipo IOException.
Create	Crea un nuevo archivo. Si el archivo existe, será sobrescrito.
Open	Abre un archivo existente.
OpenOrCreate	Abre un archivo, si existe; en caso contrario, se crea un nuevo archivo.
Truncate	Abre un archivo existente y lo trunca a cero bytes de longitud.
Append	Abre un archivo para agregarle datos al final. Si el archivo no existe, lo crea.

Acceso es un valor del tipo enumerado [FileAccess](#); puede tomar uno de los siguientes valores:

Valor	Acción
Read	Permite leer un archivo.
ReadWrite	Permite leer o escribir en el archivo.
Write	Permite escribir en el archivo.

Ejemplo 1: Escribir datos al archivo. // [CEscribirBytes.cs](#)

```
using System.IO;

public class CEscribirBytes
{
    public static void Main ( )
    {
        FileStream fs = null;
        byte[] buffer = new byte[81];
        int nbytes = 0, car;

        try
        {
            // Crea un flujo hacia el archivo texto.txt
            fs = new FileStream("texto.txt", FileMode.Create,
                               FileAccess.Write);

            Console.WriteLine("Escriba el texto que desea almacenar en el archivo:");
            while ((car = Console.Read()) != '\r' && (nbytes < buffer.Length))
            {
                buffer[nbytes] = (byte)car;
                nbytes++;
            }

            // Escribe la línea de texto en el archivo.
            fs.Write(buffer, 0, nbytes);
        }

        catch(IOException e)
        {
            Console.WriteLine("Error: " + e.Message);
            Console.WriteLine("Ruta: " + e.StackTrace);
        }

        finally
        {
            // Cierra el archivo.
            if (fs != null) fs.Close();
        }
    }
}
```

Ejemplo 2: Leer datos del archivo. // CLeerBytes.cs

```
using System.IO;

public class CLeerBytes
{
    public static void Main ( )
    {
        FileStream fe = null;
        char[] cBuffer = new char[81];
        byte[] bBuffer = new byte[81];
        int nbytes;

        try
        {
            // Crea un flujo desde el archivo texto.txt
            fe = new FileStream("texto.txt", FileMode.Open, FileAccess.Read);

            // Lee del archivo una línea de texto
            nbytes = fe.Read(bBuffer, 0, 81);

            // Crea un objeto string con el texto leído
            Array.Copy(bBuffer, cBuffer, bBuffer.Length);
            String str = new String(cBuffer, 0, nbytes);

            // Muestra el texto leído
            Console.WriteLine(str);
        }

        catch(IOException e)
        {
            Console.WriteLine("Error: " + e.Message);
            Console.WriteLine("Ruta: " + e.StackTrace);
        }

        finally
        {
            // Cierra el archivo.
            if (fe != null) fe.Close();
        }
    }
}
```

2.-Archivos de Texto con Flujos de Caracteres.

En este caso los datos se manejan por medio de flujos de caracteres, utilizando las clases [StreamWriter](#) y [StreamReader](#).

a) Escritura.

[StreamWriter](#) es una clase derivada de [TextWriter](#).

Sus constructores son:

[StreamWriter](#) ([string](#) Nombre) // Abre un nuevo flujo para escribir en un archivo especificado por nombre.

[StreamWriter](#) ([Stream](#) Flujo) // Utiliza un flujo existente para escribir.

Antes de ser escritos, los datos son convertidos automáticamente a un formato portable de 8 bits (UTF-8, UCS Transformation Format 8).

Los objetos de la clase [StreamWriter](#) poseen varios métodos, entre los que destacan:

[Write\(\)](#) ,
[WriteLine\(\)](#) y
[Flush\(\)](#) .

Además, poseen la propiedad [BaseStream](#).

b) Lectura.

[StreamReader](#) es una clase derivada de [TextReader](#).

Sus constructores son:

[StreamReader](#) ([string](#) Nombre) // Abre un nuevo flujo para leer de un archivo especificado por nombre.

[StreamReader](#) ([Stream](#) Flujo) // Utiliza un flujo existente para leer.

Algunos de los métodos más importantes de la clase [StreamReader](#) son:

[Read\(\)](#) ,
[ReadLine\(\)](#) ,
[Peek\(\)](#) y
[DiscardBufferData\(\)](#) .

Ejemplo 3: Escribir datos al archivo. // CEscribirCars.cs

```
using System.IO;

public class CEscribirCars
{
    public static void Main ( )
    {
        StreamWriter sw = null;
        String str;

        try
        {
            // Crea un flujo hacia el archivo doc.txt, si el archivo existe se
            // destruye su contenido.
            sw = new StreamWriter("doc.txt");

            Console.WriteLine("Escriba las líneas de texto para almacenar en el
            archivo.\n" + "Finalice cada línea pulsando la tecla <Entrar>.\n" +
            "Para finalizar la escritura pulse sólo la tecla <Entrar>.\n");

            // Lee una línea de la entrada estándar
            str = Console.ReadLine();

            // Mientras la cadena str no esté vacía
            while (str.Length != 0)
            {
                // Escribe la línea leída en el archivo
                sw.WriteLine(str);
                // Lee la línea siguiente
                str = Console.ReadLine();
            }

            catch(IOException e)
            {
                Console.WriteLine("Error: " + e.Message);
                Console.WriteLine("Ruta: " + e.StackTrace);
            }

            finally
            {
                // Cierra el archivo.
                if (sw != null) sw.Close();
            }
        }
    }
}
```

Ejemplo 4: Leer datos del archivo. // CLeercars.cs

```
using System.IO;

public class CLeerCars
{
    public static void Main (string[] args)
    {
        StreamReader sr = null;
        String str;

        try
        {
            // Crea un flujo desde el archivo doc.txt
            sr = new StreamReader("doc.txt");

            // Lee del archivo una línea de texto
            str = sr.ReadLine();

            // Mientras la cadena str no esté vacía
            while (str != null)
            {
                // Muestra la línea leída y lee la línea siguiente
                Console.WriteLine(str);
                str = sr.ReadLine();
            }

        }
        catch (IOException e)
        {
            Console.WriteLine("Error: " + e.Message);
            Console.WriteLine("Ruta: " + e.StackTrace);
        }

        finally
        {
            // Cierra el archivo
            if (sr != null) sr.Close();
        }
    }
}
```

b) Archivos binarios.

Cuando se requiere efectuar operaciones con datos de alguno de los tipos primitivos, tales datos deberán escribirse y leerse en formato binario.

El espacio de nombres [System.IO](#) proporciona las clases: [BinaryWriter](#) y [BinaryReader](#).

Con estas clases se podrán manipular datos de los tipos primitivos y cadenas de caracteres en formato UTF-8.

Los archivos escritos en formato binario no se pueden desplegar directamente en los dispositivos de salida estándar, como el monitor, sino que deben leerse a través de flujos de la clase [BinaryReader](#).

1) Escritura.

[BinaryWriter](#) crea flujos para escribir archivos con datos de los tipos primitivos en formato binario.

Su constructor es:

[BinaryWriter](#) (Stream Flujo)

// Requiere, como parámetro, un flujo de la clase [Stream](#) o sus derivadas.

Ejemplo:

```
FileStream fs = new FileStream ("datos.dat", FileMode.Create, FileAccess.Write);  
BinaryWriter bw = new BinaryWriter( fs );
```

Un objeto de la clase [BinaryWriter](#) actúa como filtro entre el programa y un flujo de la clase [FileStream](#).

En la siguiente tabla se describen algunos de los principales métodos y propiedades de la clase [BinaryWriter](#):

Método / Propiedad	Descripción
Write(byte)	Escribe un valor de tipo byte.
Write(byte[])	Escribe una cadena como una secuencia de bytes.
Write(char)	Escribe un valor de tipo char.
Write(char[])	Escribe una cadena como una secuencia de caracteres.
Write(short)	Escribe un valor de tipo short.
Write(int)	Escribe un valor de tipo int.
Write(long)	Escribe un valor de tipo long.
Write(Decimal)	Escribe un valor de tipo Decimal.
Write(float)	Escribe un valor de tipo float.
Write(double)	Escribe un valor de tipo double.
Write(string)	Escribe una cadena de caracteres en formato UTF-8. El primero o los dos primeros bytes especifican el número de bytes de datos escritos en la cadena.
BaseStream	Obtiene el flujo base (fs en el ejemplo mostrado).
Close	Cierra el flujo y libera los recursos adquiridos.

Flush	Limpia el buffer asociado con el flujo.
Seek	Establece el apuntador de Lectura/Escritura en el flujo.

2) Lectura.

[BinaryReader](#) crea flujos para leer archivos con datos de los tipos primitivos en formato binario, escritos por un flujo de la clase [BinaryReader](#).

Su constructor es:

[BinaryReader \(Stream Flujo\)](#)

// Requiere, como parámetro, un flujo de la clase [Stream](#) o sus derivadas.

Ejemplo:

```
FileStream fs = new FileStream("datos.dat", FileMode.Open, FileAccess.Read);
BinaryReader br = new BinaryReader(fs);
```

Un objeto de la clase [BinaryReader](#) actúa como filtro entre un flujo de la clase [FileStream](#) y el programa.

En la siguiente tabla se describen algunos de los principales métodos y propiedades de la clase [BinaryReader](#):

Método / Propiedad	Descripción
Read(byte)	Devuelve un valor de tipo byte.
Read (byte[])	Devuelve una cadena como una secuencia de bytes.
Read (char)	Devuelve un valor de tipo char.
Read (char[])	Devuelve una cadena como una secuencia de caracteres.
Read (short)	Devuelve un valor de tipo short.
Read (int)	Devuelve un valor de tipo int.
Read (long)	Devuelve un valor de tipo long.
Read (Decimal)	Devuelve un valor de tipo Decimal.
Read (float)	Devuelve un valor de tipo float.
Read (double)	Devuelve un valor de tipo double.
Read (string)	Devuelve una cadena de caracteres en formato UTF-8. El primero o los dos primeros bytes especifican el número de bytes de datos que serán leídos.
BaseStream	Obtiene el flujo base (fs en el ejemplo mostrado).
Close	Cierra el flujo y libera los recursos adquiridos.
Flush	Limpia el buffer asociado con el flujo.
PeekChar	Obtiene el siguiente carácter, sin extraerlo.

Ejemplo 5: Escribir y leer datos a un archivo binario. // [ArchivoBinario.cs](#)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.IO;

namespace EjemploArchivoBinario
{
    class ArchivoBinarioEmpleados
    {
        //declaracion de flujos
        BinaryWriter bw = null; //flujo salida - escritura de datos
        BinaryReader br = null; //flujo entrada - lectura de datos

        //campos de la clase
        string Nombre, Direccion;
        long Telefono;
        int NumEmp, DiasTrabajados;
        float SalarioDiario;

        public void CrearArchivo(string Archivo)
        {
            //variable local método
            char resp;

            try
            {
                //creación del flujo para escribir datos al archivo
                bw = new BinaryWriter(new FileStream(Archivo,
                FileMode.Create, FileAccess.Write));

                //captura de datos
                do
                {
                    Console.Clear();
                    Console.Write("Numero del Empleado: "); NumEmp =
                    Int32.Parse(Console.ReadLine());
                    Console.Write("Nombre del Empleado: "); Nombre =
                    Console.ReadLine();
                    Console.Write("Dirección del Empleado: "); Direccion =
                    Console.ReadLine();
                    Console.Write("Teléfono del Empleado: "); Telefono =
                    Int64.Parse(Console.ReadLine());
                    Console.Write("Dias Trabajados del Empleado: ");
                    DiasTrabajados = Int32.Parse(Console.ReadLine());
                    Console.Write("Salario Diario del Empleado: ");
                    SalarioDiario = Single.Parse(Console.ReadLine());

                    //escribe los datos al archivo
                    bw.Write(NumEmp);
                    bw.Write(Nombre);
                    bw.Write(Direccion);
                    bw.Write(Telefono);
                    bw.Write(DiasTrabajados);
                    bw.Write(SalarioDiario);
                } while (resp != 'n');
            }
            catch { }
        }
    }
}
```

```

        Console.WriteLine("\n\nDeseas Almacenar otro Registro (s/n)?");
    };

    resp = Char.Parse(Console.ReadLine());
} while ((resp == 's') || (resp == 'S'));
}

catch (IOException e)
{
    Console.WriteLine("\nError : " + e.Message);
    Console.WriteLine("\nRuta : " + e.StackTrace);
}

finally
{
    if (bw != null) bw.Close(); //cierra el flujo - escritura
    Console.WriteLine("\nPresione <enter> para terminar la Escritura de Datos y regresar al Menu.");
    Console.ReadKey();
}
}

public void MostrarArchivo(string Archivo)
{
    try
    {
        //verifica si existe el archivo
        if (File.Exists(Archivo))
        {
            //creación flujo para leer datos del archivo
            br = new BinaryReader(new FileStream(Archivo,
FileMode.Open, FileAccess.Read));

            //despliegue de datos en pantalla
            Console.Clear();
            do
            {
                //lectura de registros mientras no llegue a EndOfFile
                NumEmp = br.ReadInt32();
                Nombre = br.ReadString();
                Direccion = br.ReadString();
                Telefono = br.ReadInt64();
                DiasTrabajados = br.ReadInt32();
                SalarioDiario = br.ReadSingle();

                //muestra los datos
                Console.WriteLine("Numero del Empleado      : " +
NumEmp);
                Console.WriteLine("Nombre del Empleado      : " +
Nombre);
                Console.WriteLine("Dirección del Empleado    : " +
Direccion);
                Console.WriteLine("Teléfono del Empleado    : " +
Telefono);
                Console.WriteLine("Dias Trabajados del Empleado : " +
DiasTrabajados);
            }
            while (br.Peek() != -1);
        }
    }
    catch (Exception e)
    {
        Console.WriteLine("\nError : " + e.Message);
    }
}

```

```

        Console.WriteLine("Salario Diario del Empleado :
{0:C} ",SalarioDiario);
        Console.WriteLine("SUELDO TOTAL del Empleado : {0:C}
", (DiasTrabajados*SalarioDiario));
        Console.WriteLine("\n");
    } while (true);
    }
    else
    {
        Console.Clear();
        Console.WriteLine("\n\nEl Archivo " + Archivo + " No
Existe en el Disco!!");
        Console.Write("\nPresione <enter> para Continuar...");
        Console.ReadKey();
    }
}

catch (EndOfStreamException)
{
    Console.WriteLine("\n\nFin del Listado de Empleados");
    Console.Write("\nPresione <enter> para Continuar...");
    Console.ReadKey();
}

finally
{
    if (br != null) br.Close(); //cierra flujo
    Console.Write("\nPresione <enter> para terminar la Lectura
de Datos y regresar al Menu.");
    Console.ReadKey();
}
}

static void Main(string[] args)
{
    //declaración variables auxiliares
    string Arch = null;
    int opcion;

    //creación del objeto
    ArchivoBinarioEmpleados A1 = new ArchivoBinarioEmpleados();

    //Menu de Opciones
    do
    {
        Console.Clear();
        Console.WriteLine("\n*** ARCHIVO BINARIO EMPLEADOS***");
        Console.WriteLine("1.- Creación de un Archivo.");
        Console.WriteLine("2.- Lectura de un Archivo.");
        Console.WriteLine("3.- Salida del Programa.");
        Console.Write("\nQue opción deseas: ");
        opcion = Int16.Parse(Console.ReadLine());

        switch (opcion)
        {
            case 1:
                //bloque de escritura

```

```

        try
        {
            //captura nombre archivo
            Console.Write("\nAlimenta el Nombre del Archivo a
Crear: "); Arch = Console.ReadLine();

            //verifica si existe el archivo
            char resp = 's';
            if (File.Exists(Arch))
            {
                Console.Write("\nEl Archivo Existe!!, Deseas
Sobreescribirlo (s/n) ? ");
                resp = Char.Parse(Console.ReadLine());
            }
            if ((resp == 's') || (resp == 'S'))
                Al.CrearArchivo(Arch);
        }
        catch (IOException e)
        {
            Console.WriteLine("\nError : " + e.Message);
            Console.WriteLine("\nRuta : " + e.StackTrace);
        }
        break;

    case 2:
        //bloque de lectura
        try
        {
            //captura nombre archivo
            Console.Write("\nAlimenta el Nombre del Archivo
que deseas Leer: "); Arch = Console.ReadLine();
            Al.MostrarArchivo(Arch);
        }
        catch (IOException e)
        {
            Console.WriteLine("\nError : " + e.Message);
            Console.WriteLine("\nRuta : " + e.StackTrace);
        }
        break;

    case 3:
        Console.Write("\nPresione <enter> para Salir del
Programa.");
        Console.ReadKey();
        break;

    default:
        Console.Write("\nEsa Opción No Existe!!, Presione
<enter> para Continuar...");
        Console.ReadKey();
        break;
    }
} while (opción != 3);
}
}

```

Acceso Aleatorio.

Cuando se abre un archivo, se puede acceder a sus registros de manera secuencial o de manera aleatoria.

El acceso aleatorio consiste en posicionar el apuntador de archivo en una localidad específica del archivo. Este tipo de acceso es necesario cuando se requiere modificar alguno de los campos de un registro. En lugar de leer desde el primero hasta el registro elegido, el apuntador se posiciona, en una especie de salto, hasta dicho registro.

El salto se hace con base al conteo de los bytes existentes entre una posición inicial y el primer byte del registro elegido.

Para el acceso aleatorio a los registros de un archivo, la clase `FileStream` implementa las propiedades `Position` y `Length`, así como el método `Seek()`.

`Position` devuelve la posición actual, en bytes, del apuntador de archivo.

El apuntador de archivo marca el byte donde se hará la siguiente operación de lectura o de escritura.

La declaración de la propiedad `Position` es: `public long Position;`

La propiedad `Length` devuelve la longitud del archivo en bytes: `public long Length ;`

El método `Seek()` mueve el apuntador de archivo a una ubicación localizada `desp` bytes, a partir de la posición `pos` del archivo.

La sintaxis para `Seek()` es: `public long Seek( long desp , SeekOrigin pos );`

El desplazamiento `desp` puede ser positivo o negativo.

La posición `pos` puede tomar uno de los siguientes valores del tipo enumerado `SetOrigin`.

`Begin` El inicio del archivo.

`Current` Posición actual del apuntador de archivo.

`End` El final del archivo.

Ejemplos:

1.- Suponiendo un flujo `fs` de la clase `FileStream`.

```
fs.Seek( desp , SeekOrigin. Begin);
```

```
fs.Seek( desp , SeekOrigin. Current);
```

```
fs.Seek( -desp , SeekOrigin. End);
```

El valor almacenado en `desp` debe calcularse en base al tamaño de registro del archivo.

2.- Suponiendo un flujo `br` de la clase `BinaryReader`.

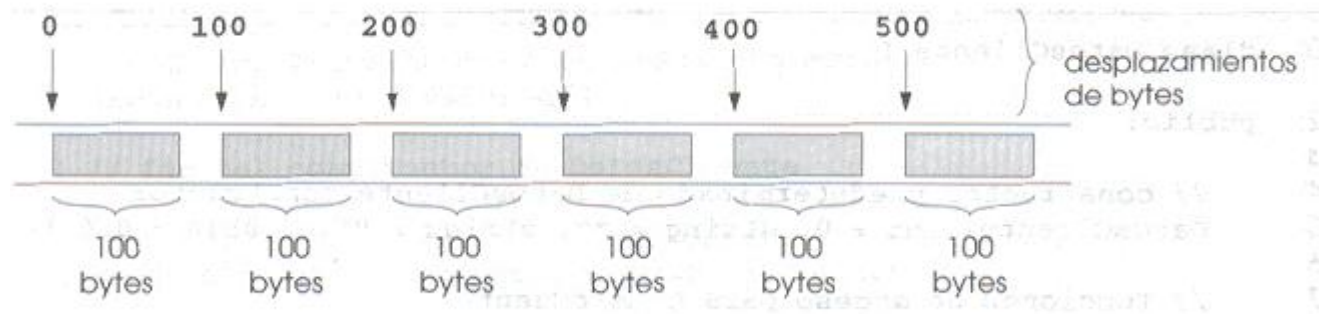
```
FileStream fs = new FileStream("datos", FileMode.Open, FileAccess.Read);
```

```
BinaryReader br = new BinaryReader( fs );
```

Debido a que el flujo `br` no soporta directamente el método `Seek( )`, tiene que acceder a él a través de su propiedad `BaseStream`, como se muestra a continuación:

```
br.BaseStream.Seek (desp, SeekOrigin.Current);
```

Para facilitar el cálculo del desplazamiento `desp`, es conveniente que todos los registros tengan la misma longitud.



En la figura anterior se muestra un archivo con 6 registros (numerados de 0 a 5) con 100 bytes de longitud cada uno.

El desplazamiento para acceder al tercer registro se calcula de la siguiente manera:

`desp = Numero de registro * tamaño de registro`

`desp = 2*100`

Es difícil recordar el número de registro al que se necesita acceder, por lo que se hace necesario establecer una relación entre el número de registro y el valor almacenado en alguno de los campos del registro.

Por ejemplo, si cada registro tiene un campo denominado `clave`, de tipo entero en el rango de 1 a N (que corresponde al número de control de un empleado), la relación entre la `clave` y el número de registro es:

`Número de registro = clave -1`

Así, el cálculo del desplazamiento para el empleado con la clave 4 se efectúa de la siguiente manera:

`desp = (4-1) * 100 = 300`

Esto obliga a que, cuando se introduzcan los datos de un nuevo empleado, el número de su clave deba ser igual al número de registro +1.

El número de registro que sigue es igual al número de registros existentes.

6.4 – Manejo de objetos persistentes.

¿Qué es Persistencia?

Es la propiedad de un objeto por la que su existencia trasciende el tiempo es decir, el objeto continúa existiendo después de que su creador deja de existir y/o espacio.

Dicho de otra manera, la persistencia es la acción de mantener la información del objeto de una forma permanente (guardarla), pero también debe de poder recuperarse dicha información para que pueda ser utilizada nuevamente.

La persistencia es el mecanismo que se usa para mantener información almacenada.

Para la persistencia los objetos podrían clasificarse en dos tipos objetos transitorios y objetos persistentes.

Transitorios: Son aquellos que su tiempo de vida depende del espacio del proceso que lo creo.

Persistentes: Son aquellos que su estado es almacenado en un medio temporal para su posterior reconstrucción y utilización, por lo cuál el objeto no depende del proceso que lo creo.

Un ejemplo de la persistencia es aquel objeto que se crea para luego ser guardado en la base de datos.

Las clases persistentes son clases en una aplicación que implementan las entidades del problema empresarial (por ejemplo, Customer y Order en una aplicación de comercio electrónico). No se considera que todas las instancias de una clase persistente estén en estado persistente. Por ejemplo, una instancia puede ser transitoria o separada.

Hibernate funciona mejor si estas clases siguen algunas reglas simples, también conocidas como el modelo de programación POJO (Plain Old Java Object). Sin embargo, ninguna de estas reglas son requerimientos rígidos. De hecho, Hibernate3 asume muy poco acerca de la naturaleza de sus objetos persistentes. Puede expresar un modelo de dominio en otras formas (por ejemplo, utilizando árboles de instancias de Map).

La mayoría de aplicaciones Java requieren una clase persistente que represente a los felinos. Por ejemplo:

```
package eg;
import java.util.Set;
import java.util.Date;

public class Cat {
    private Long id; // identifier
    private Date birthdate;
    private Color color;
    private char sex;
    private float weight;
    private int litterId;
    private Cat mother;
    private Set kittens = new HashSet();
```

```

private void setId(Long id) {
    this.id=id;
}
public Long getId() {
    return id;
}
void setBirthdate(Date date) {
    birthdate = date;
}
public Date getBirthdate() {
    return birthdate;
}
void setWeight(float weight) {
    this.weight = weight;
}
public float getWeight() {
    return weight;
}
public Color getColor() {
    return color;
}
void setColor(Color color) {
    this.color = color;
}
void setSex(char sex) {
    this.sex=sex;
}
public char getSex() {
    return sex;
}
void setLitterId(int id) {
    this.litterId = id;
}
public int getLitterId() {
    return litterId;
}
void setMother(Cat mother) {
    this.mother = mother;
}
public Cat getMother() {
    return mother;
}
void setKittens(Set kittens) {
    this.kittens = kittens;
}
public Set getKittens() {
    return kittens;
}
public void addKitten(Cat kitten) {
    kitten.setMother(this);
    kitten.setLitterId( kittens.size() );
    kittens.add(kitten);
}
}

```

Vamos a explorar en mayor detalle las cuatro reglas principales de las clases persistentes:

Implemente un constructor sin argumentos

Cat tiene un constructor sin argumentos. Todas las clases persistentes deben tener un constructor predeterminado (el cual puede ser no-público) de modo que Hibernate pueda instanciarlas usando `Constructor.newInstance()`. Le recomendamos contar con un constructor por defecto con al menos una visibilidad de *paquete* para la generación de proxies en tiempo de ejecución en Hibernate.

Proporcione una propiedad identificadora (opcional)

Cat tiene una propiedad llamada `id`. Esta propiedad mapea a la columna de la llave principal de la tabla de la base de datos. La propiedad podría llamarse de cualquier manera y su tipo podría haber sido cualquier tipo primitivo, cualquier tipo de "wrapper" primitivo, `java.lang.String` o `java.util.Date`. Si su tabla de base de datos heredada tiene claves compuestas, puede utilizar una clase definida por el usuario con propiedades de estos tipos

La propiedad identificadora es estrictamente opcional. Puede olvidarla y dejar que Hibernate siga internamente la pista de los identificadores del objeto. Sin embargo, no recomendamos que esto suceda.

De hecho, algunas funcionalidades se encuentran disponibles solamente para clases que declaran una propiedad identificadora:

`Session.saveOrUpdate()`

`Session.merge()`

Le recomendamos que declare propiedades identificadoras nombradas-consistentemente en clases persistentes. y que utilice un tipo nutable (por ejemplo, no primitivo).

Prefiera las clases no finales

Un aspecto central de Hibernate, los proxies, dependen de que las clases persistentes sean no finales o de la implementación de una interfaz que declare todos los métodos públicos.

Con Hibernate puede persistir las clases finales que no implementen una interfaz. Sin embargo, no podrá utilizar proxies para recuperación perezosa de asociaciones, lo cual limitará sus opciones para afinar el rendimiento.

También debe evitar el declarar métodos `public final` en las clases no-finales. Si quiere utilizar una clase con un método `public final`, debe deshabilitar explícitamente el uso de proxies estableciendo `lazy="false"`.

Declare métodos de acceso y de modificación para los campos persistentes

Cat declara métodos de acceso para todos sus campos persistentes. Muchas otras herramientas ORM persisten directamente variables de instancia. Es mejor proporcionar una indirección entre el esquema relacional y las estructuras de datos internos de la clase. Por defecto, Hibernate persiste las propiedades del estilo JavaBeans, y reconoce los nombres de método de la forma `getFoo`, `isFoo` y `setFoo`. De ser necesario, puede cambiarse al acceso directo a campos para propiedades específicas.

No es necesario declarar públicas las propiedades. Hibernate puede persistir una propiedad con un par `get / set` `protected` o `private`.

REFERENCIAS BIBLIOGRAFICAS UNIDAD 6

Definición:

[https://es.wikipedia.org/wiki/Archivo_\(informática\)](https://es.wikipedia.org/wiki/Archivo_(informática))

www.alegsa.com.ar › [Alegsa.com.ar](http://www.alegsa.com.ar) › Diccionario tecnología › Sistemas de archivos

Clasificación de archivos:

www.tiposde.org/informatica/105-tipos-de-archivos/

https://es.wikipedia.org/wiki/Sistema_de_archivos

Operaciones básicas:

www.itslr.edu.mx/archivos2013/TPM/temas/s2u6.html

<https://prezi.com/unpgnffj8-hb/62-operaciones-basicas-en-archivos-texto-y-binario/>

Tipos de acceso:

<http://www.itslr.edu.mx/archivos2013/TPM/temas/s2u6.html>

<http://tareaprogr.blogspot.com/2014/06/43-tipos-de-acceso-archivos.html>

Manejo de objetos persistentes:

<https://styde.net/concurrencia-y-persistencia-en-programacion-orientada-a-objetos/>

<http://p-o-o-itsav.blogspot.mx/2012/02/64-manejo-de-objetos-persistentes.html>

<https://si.ua.es/es/documentacion/...net/.../tema-9-persistencia-y-objetos-dinamicos.pdf>