**El lenguaje de programación C#:**

* Tiempo de compilación: Se compila a lenguaje intermedio.
* Tiempo de diseño:
* Tiempo de ejecución: Se compila a lenguaje máquina/nativo

**Common Type System (CTS)**

* Define un conjunto común de “tipos” de datos orientados a objetos.
* Todo lenguaje de programación .NET debe implementar los tipos definidos por el CTS.
* Todo tipo hereda directa o indirectamente del tipo System.Object.
* El CTS define tipos de VALOR y de REFERENCIA

**Tipos de Datos**

* Las variables escalares son constantes o variable que contiene un dato atómico y unidimensional.
* Las variables no escalares son array (vector), lista y objeto, que pueden tener almacenado en su estructura más de un valor.
* Los valores predeterminados son:
* Enteros: 0 (cero)
* Punto flotante: 0 (cero)
* Lógicos: False
* Referencias: Null

**Conversiones**

* implícitas: No interviene el programador
* Explicitas: Interviene el programador ya que puede haber perdida de datos.

**Entry Point**

El punto de entrada para los programas en C# es la función Main.

* Static: Es un modificador que permite ejecutar un método sin tener que instanciar una variable (sin crear un objeto). El método Main () debe ser estático.
* Void: Indica el tipo de valor del retorno del método Main(). No necesariamente tiene que ser void
* String[] args: Es un array de tipo string que puede recibir el método Main() como parámetro. Este parámetro es opcional.

**Console**

* Es una Clase publica y estática.
* Representa la entrada, salida y errores de streams para aplicaciones de consola.
* Es miembro del NameSpace System.

**Programación Orientada a Objetos**

* Es una manera de construir soft basada en un nuevo paradigma.
* Propone resolver problemas de la realidad a través de identificar objetos y relaciones de colaboración entre ellos.
* El objeto y el mensaje son sus elementos fundamentales.
* Sus pilares son
  + **Abstracción:** Ignorancia selectiva, decide que es lo importante y que no lo es (Se enfoca en lo que es importante e ignora lo que no lo es). Utiliza el encapsulamiento para reforzar la abstracción.
  + **Encapsulamiento:** Es la capacidad del objeto de responder a peticiones a través de sus métodos o propiedades sin la necesidad de exponer los medios utilizados para llegar a brindar resultados.
  + **Herencia:** Consiste en que una clase puede heredar sus variables y métodos a varias subclases. Hereda la implementación, va de la generalización a la especialización (Clase base o padre a derivada o hija). La relación entre clases es del tipo “Es de un tipo de”.
  + **Polimorfismo:** Es la capacidad de que más de un objeto puedan crearse usando la misma clase base para lograr objetos diferentes. Se resuelve en tiempo de ejecución, la definición de un método reside en la clase base o padre y la implementación de este reside en la clase derivada o hija.

**Clase**

Una clase es la descripción de un conjunto de objetos, se puede decir que una clase es una abstracciones de objetos.

* Se clasifican en base a comportamientos y atributos comunes.
* A partir de la clasificación se crea un vocabulario.
* Es una abstracción de un objeto.
* Una clase es una clasificación.

**Namespaces**

* Es una agrupación lógica de clases y otros elementos
* Toda clase está dentro de una namespace
* Proporciona un marco de trabajo jerárquico sobre el cual se construye y se organiza todo el código
* Su función principal es la de organizar el código para reducir los conflictos entre nombres.
* Esto hace posible usar en el mismo programa componentes de distintas procedencias.

**Directivas**

Son elementos que permite a un programa identificar los namespaces que se usarán en el mismo, y permite el uso de miembros de un namespace particular sin tener que especificar un nombre completamente cualificado.

C# posee 2 directivas de namespaces:

* Using: Permite la especificación de una llamada a un método sin el uso obligatorio de un nombre completamente cualificado
* Alias: permite usar un nombre distinto para un namespaces, generalmente se usa para abreviar los nombres largos.

**Objetos**

Los objetos son instancias de una clase. Cuando creamos una instancia, tenemos que especificar la clase a partir de la cual se creará.

* Los objetos son clases instanciadas que se crean en tiempo de ejecución, **poseen comportamiento y Estados.**
* Para acceder a los métodos o atributos se utiliza el punto, para crear un objeto se necesita la palabra reservada new

Ciclo de vida de un objeto:

* Creación del objeto:
  + Se usa new para asignar memoria, y se usa un constructor para inicializar un objeto en esa memoria. **Las dos cosas no pasan en la misma acción.**
* Destrucción del objeto:
  + **Se pierde la referencia de memoria ya sea por finalización del programa**, cambio o eliminación de la variable. El garbage colector liberará memoria cuando lo crea necesario.

**Garbage Colector**

Será el encargado de liberar memoria cada vez que creamos un objeto. El CLR asigna memoria desde la porción gestionada y eventualmente liberará la memoria de objetos sin referencia.

**La memoria y tipo de datos**

El CLR administra dos segmentos de memoria, los cuales son utilizados de distintas formas a lo largo del ciclo de vida de una aplicación.

**El STACK o PILA es una sección de memoria que almacena los tipos de valor**, son llamados así porque tanto su referencia como su valor se encuentra en la misma posición de memoria.

* En el STACK se almacena una referencia al contenido de la variable.

**El HEAP es una sección de memoria que almacena los tipos de referencia**. Son llamados así, porque su almacenamiento se encuentra dividido.

* En el HEAP Se guarda el valor propiamente dicho de la variable.

El tiempo de vida de una **variable** local está vinculado al ámbito en el que este se declara. El tiempo de vida de un **objeto** dinámico no está vinculado su ámbito.

**Variables:**

Creación y destrucción deterministas

**Una variable local** se crea al momento de declararla y se destruye al final del ámbito en el que está declarada. El punto inicial y el punto final de la vida del valor son deterministas, es decir, tienen lugar en momentos conocidos y fijos.

Tiempos de vida muy cortos

Un valor se declara en alguna parte de un método, y no puede existir más allá de una llamada al método, cuando un método devuelve un valor lo que se devuelve es una copia del valor.

**Objetos**

Destrucción no determinista

**Un objeto** aparece cuando se crea, pero a diferencia de un valor, no se destruye al final del ámbito en el que se crea. La creación de un objeto es determinista, pero no así su destrucción. No es posible controlar exactamente cuándo se destruye y libera memoria para un objeto.

* Son tipos por referencia y se almacenan en la sección de memoria conocida como heap.

Tiempos de vida más largos

Puesto que el tiempo de vida de un objeto no está vinculado al método que lo crea, un objeto puede existir mucho más allá de una llamada al método.

* El tiempo de vida de un objeto no está vinculado al ámbito en el que se crea. Los objetos se inicializan en memoria del Managed Heap mediante el operador new.

**Destrucción de Objetos**

**No es posible destruir objetos de forma explícita,** esto se debe a que una función de eliminación explícita es una importante fuente de errores en otros lenguajes.

Los objetos se destruyen por Garabage Collector

* Busca objetos inalcanzables y los destruye.
* Los convierte de nuevo en memoria binaria no utilizada.
* Normalmente lo hace cuando empieza a faltar memoria o cuando finaliza la aplicación.

**Constantes**

* Una constante es otro tipo de campo.
* Contiene un valor que se asigna cuando se compila el programa y nunca cambia.
* Las constantes se declaran con la palabra clave const; son útiles para que el código sea más legible.

**Constructores**

* Los constructores **son métodos especiales** que se utilizan para inicializar objetos al momento de su creación.
* En C#, la única forma de crear un objeto es mediante el uso de la palabra reservada new para adquirir y asignar memoria.
* Aunque no se escriba ningún constructor, existe uno por defecto que se usa cuando se crea un objeto a partir de un tipo referencia.
* Los constructores llevan el mismo nombre de la clase.
* Los Constructores Sólo inicializan el estado del objeto con los valores necesarios para asegurar su correcto funcionamiento. **No realizan ninguna función en la adquisición de memoria.**

**New**

* Lo único que **new** hace es adquirir memoria binaria sin inicializar, **no participa en la inicialización**
* Aunque **new** y los constructores de instancia realizan tareas independientes, un programador no puede emplearlos por separado. De esta forma, C# contribuye a garantizar que la memoria está siempre configurada para un valor válido antes de que se lea (a esto se le llama *asignación definida*).

Hay dos tipos de constructores:

* Constructores de instancia: que inicializan objetos (atributos NO estáticos).
* Constructores estáticos: que son los que inicializan clases (atributos estáticos).

Constructores por defecto:

* Acceso público.
* No tiene tipo de retorno (ni siquiera void).
* No recibe ningún argumento.
* Inicializa todos los campos a cero, false o null.

Constructores Estaticos:

* Son los encargados de inicializar clases.
* Sólo inicializará los atributos estáticos.
* No debe llevar modificadores de acceso.
* Utilizan la palabra reservada static.
* No pueden recibir parámetros.

**Sobrecarga de Metodos**

Los métodos no pueden tener el mismo nombre que otros elementos en una misma clase (atributos, propiedades, etc), sin embargo, dos o más métodos en una clase si pueden compartir el nombre. A esto se lo denomina sobrecarga.

Los métodos se sobrecargan cambiando el número, el tipo y el orden de los parámetros (se cambia la firma del método), el compilador de C# distingue métodos sobrecargados comparando la lista de parámetros.

* Las firmas de los métodos deben ser únicas dentro de una clase.
* Forman la definición de la firma de un método:
  + Nombre del método.
  + Tipo de parámetros.
  + Cantidad de parámetros.
  + Modificador de parámetro (out o ref)
* No afectan la firma de un método:
  + Nombres de parámetros.
  + Tipo de retorno del método.

Uso de métodos Sobrecargados:

* Si hay métodos similares que requieren parámetros diferentes.
* Si se quiere añadir funcionalidad al código existente.
* Son difíciles de depurar.
* Son difíciles de mantener.

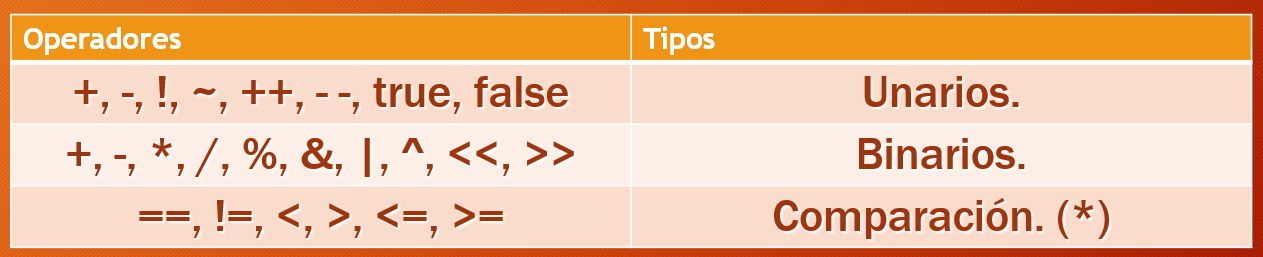
Sobrecarga de constructores:

* Al igual que los métodos, los constructores también se pueden sobrecargar.
* Las normas para hacerlo son las mismas.
* Se suele hacer cuando se quiere dar la posibilidad de instanciar objetos de formas diferentes.

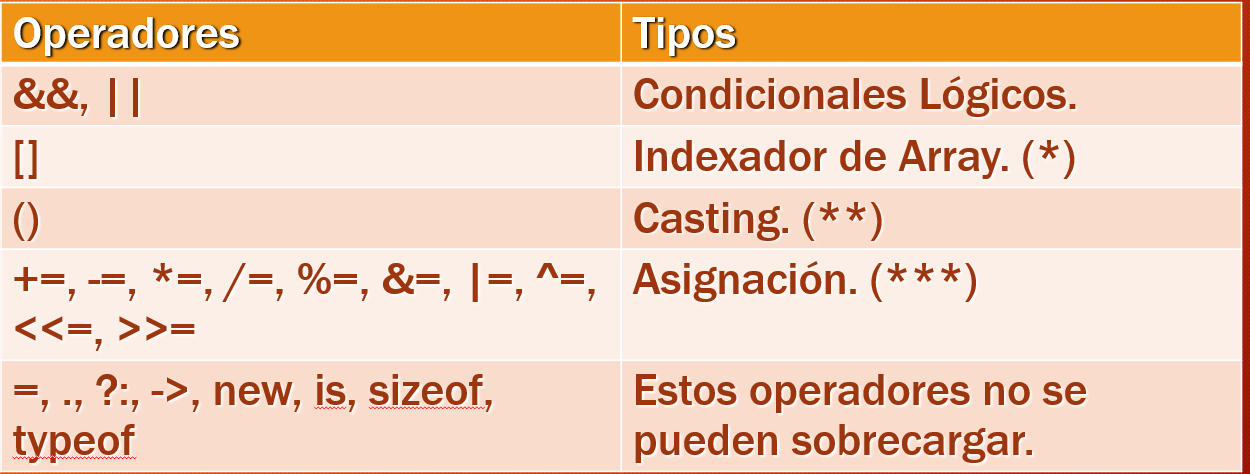
**Sobrecarga de Operadores**

Sobrecargar un operador consiste en modificar su comportamiento cuando este se utiliza con una determinada clase.

* El modificador de acceso no podrá ser de un ámbito mayor que el de la clase.
* Los operadores de Comparación, si son sobrecargados, se deben sobrecargar en pares; es decir, si se sobrecarga el operador igual (==), se deberá sobrecargar el operador distinto (!=).
* Sobrecargables:



* No Sobrecargables:



Operadores de conversión:

* Las conversiones **definidas permiten hacer compatibles tipos que antes no lo eran.**
* Los operadores de conversión **pueden ser implícitos o explícitos.**
* Los operadores de conversión explícitos son muy usados cuando se quiere que los usuarios estén conscientes que una conversión se llevará a cabo.

**Formularios**

Los formularios son objetos que exponen propiedades, métodos que definen su comportamiento y **eventos** que definen la interacción con el usuario.

* Al momento de diseñar un formulario, el diseñador de Visual Studio escribe de forma automática el código que describe a cada uno de los controles y al propio formulario.
* El concepto de Partial Class, que se incorpora en .NET 2.0, permite separar el código de una clase en dos archivos fuentes diferentes.
* El diseñador de formularios utiliza esta técnica para escribir en un archivo aparte todo el código que él mismo genera.
* Esto permite organizar más claramente el código, manteniendo separada la lógica de la aplicación en un archivo diferente.

**Arrays**

Un array puede ser unidimensional, multidimensional o anidado (jagged). El valor por defecto de Array de elementos numéricos (value types) se establece en cero, mientras que los objetos (reference types) se establece en null. Un Array anidado es un Array de Arrays, por lo tanto, al ser tipos por referencia se inicializan en null.

* Los Arrays en C# son base-cero: un Array con ‘n’ elementos está indexado de 0 a n-1.
* Los elementos de un Array pueden ser de cualquier tipo, incluso de tipo Array (Clase).
* Los Arrays son reference type, heredan de la clase abstracta System.Array.
* Implementan la interfaz IEnumerable por lo tanto se pueden iterar usando foreach.
* Aquí los Arrays son objetos derivados de la clase System.Array.
* Por lo tanto, cuando se declara un Array en C# este aún no se habrá creado, es decir, no se habrá reservado aún memoria para él.
* En consecuencia, los Arrays de C# son todos dinámicos, y antes de poder usarlos habrá que instanciarlos, como si fuera cualquier otro objeto.

**Colecciones**

Existen dos formas de agrupar objetos: mediante la creación de matrices de objetos y mediante la creación de colecciones de objetos. Las matrices son muy útiles para crear y trabajar con un número fijo de objetos fuertemente tipados.

Las colecciones proporcionan un método más flexible para trabajar con grupos de objetos.

A diferencia de las matrices, el grupo de objetos con el que trabaja puede aumentar y reducirse dinámicamente a medida que cambian las necesidades de la aplicación.

Una colección es una clase, de modo que antes de poder agregar elementos a una nueva colección, debe declararla.

Una colección genérica cumple la seguridad de tipos para que ningún otro tipo de datos se pueda agregar a ella. Cuando se recupera un elemento de una colección genérica, no tiene que determinar su tipo de datos ni convertirlo.

Colecciones Genéricas:

* Se puede crear una colección genérica utilizando una de las clases en el espacio de nombres System.Collections.Generic.
* Una colección genérica es útil cuando todos los elementos de la colección tienen el mismo tipo de datos.
* Una colección genérica cumple el tipado fuerte al permitir agregar sólo el tipo de datos deseado.
* Tipo:
  + **Dictionary**: Representa una colección de pares de clave y valor que **se organizan por claves.**
  + **List**: Representa una lista de objetos que pueden ser **obtenidos mediante un índice**. Proporciona métodos para buscar, ordenar y modificar listas.
  + **Queue**: Representa una colección de objetos con el orden primero en entrar, primero en salir (FIFO).
  + **SortedList**: Representa una colección de pares de clave y valor que se ordenan por claves según la implementación de la interfaz IComparer<T> asociada.
  + **Stack**: Representa una colección de objetos con el orden último en entrar, primero en salir (LIFO).

Colecciones No Genéricas:

* Son las incluidas en el espacio de nombres System.Collections.
* Estas no almacenan los elementos como objetos de un tipo específico, sino como objetos de tipo Object.
* Siempre que sea posible, se deberían utilizar las colecciones genéricas de otros tipos en lugar de estas.
* Tipo:
  + **ArrayList**: Representa una matriz de objetos cuyo tamaño aumenta dinámicamente según sea necesario.
  + **Hashtable**: Representa una colección de pares de clave y valor que se organizan por código hash de la clave.
  + **Queue**: Representa una colección de objetos con el orden primero en entrar, primero en salir (FIFO).
  + **Stack**: Representa una colección de objetos con el orden último en entrar, primero en salir (LIFO).

No todas las colecciones tienen las mismas propiedades, las colas y pilas no pueden ordenarse en si mismas así como tampoco se pueden serializar (pasar a archivo).

**Encapsulamiento**

Se denomina encapsulamiento al ocultamiento del estado, es decir, de los datos miembro de un objeto de manera que solo se pueda cambiar mediante las operaciones definidas para ese objeto. Cada objeto está aislado del exterior, protegiendo a los datos asociados de un objeto contra su modificación por quien no tenga derecho a acceder a ellos.

La encapsulación se encarga de mantener ocultos los procesos internos que necesita para hacer lo que sea que haga, dándole al programador acceso sólo a lo que necesita.

Niveles de Encapsulamiento

* Público: todos pueden acceder a los datos o métodos de una clase que se definen con este nivel, este es el nivel más bajo, esto es lo que tu quieres que la parte externa vea.
* Protegido: podemos decir que estás no son de acceso público, solamente son accesibles dentro de su clase y por subclases.
* Privado: en este nivel se puede declarar miembros accesibles sólo para la propia clase.

Propiedades

Una propiedad es un miembro que proporciona un mecanismo flexible para leer, escribir o calcular el valor de un campo. Las propiedades se pueden usar como si fueran miembros de datos públicos, pero en realidad son métodos especiales denominados descriptores de acceso.

Las propiedades permiten que una clase exponga una manera pública de obtener y establecer valores, a la vez que se oculta el código de implementación o validación.

Indexadores

* Los indexadores permiten a la instancia de una clase ser indexada tal cómo un array.
* La declaración de un indexador luce cómo una propiedad, sólo que:
* Este recibe parámetros.
* La palabra clave **this** se utiliza para su definición
* No es necesario indexarlos sólo con un entero.

Enumaerados

Son un conjunto propio de constantes con nombre.

Estos tipos de datos permiten declarar un conjunto de nombres u otros valores literales que definen todos los valores posibles que se pueden asignar a una variable, Por dentro, estas constantes están asociadas con el tipo de dato **int**.

Normalmente es mejor definir un **enum** directamente dentro de un espacio de nombres para que todas las clases del espacio de nombres puedan acceder a él con igual comodidad. Sin embargo, un **enum** también se puede anidar dentro de una clase o **struct**.

**Herencia**

Es una relación entre clases en la cual una clase comparte la estructura y comportamiento definido en otra clase, cada clase que hereda de otra posee los atributos de la clase base además de los propios y soporta todos o algunos de los métodos de la clase base. Una subclase hereda de una clase base.

El propósito principal de la herencia es el de organizar mejor las clases que componen una determinada realidad, y poder agruparlas en función de atributos y comportamientos comunes. A la vez que cada una se especializa según sus particularidades. La herencia permite crear nuevas clases a partir de otras ya existentes (en lugar de crearlas partiendo de cero).

La clase en la que está basada la nueva clase se la conoce como *clase base o* padre, mientras que la clase hija se conoce como *clase derivada*.

Tipos de Herencia

* Herencia Simple:
  + Una clase derivada puede heredar sólo de una clase base (los lenguajes .NET soportan este tipo de herencia)
* Herencia Múltiple:
  + Una clase derivada puede heredar de una o más clases base (C++ es un ejemplo de lenguaje que soporta este tipo de herencia).

Herencia de clase derivada:

Una clase derivada hereda todo de su clase base, excepto los constructores, los miembros públicos de la clase base se convierten implícitamente en miembros públicos de la clase derivada.

Sólo los miembros de la clase base tienen acceso a los miembros privados de esta clase, aunque la clase derivada también los hereda. Una clase derivada no puede ser más accesible que su clase base. Por ejemplo, no es posible derivar una clase pública de una clase privada…

Protected:

* El significado del modificador de acceso protected depende de la relación entre la clase que tiene el modificador y la clase que intenta acceder a los miembros que usan el modificador.
* Para una clase derivada, la palabra reservada protected es equivalente a la palabra public.
* Entre dos clases que no tengan una relación base-derivada, por el contrario, los miembros protegidos de una clase se comportan como miembros privados para la otra clase.

Miembros Heredados:

Cuando una clase derivada hereda un miembro protected, ese miembro también es implícitamente un miembro protegido de la clase derivada. Esto significa que todas clases que deriven directa o indirectamente de la clase base pueden acceder a los miembros protegidos.

Los métodos de una clase derivada sólo tienen acceso a sus propios miembros heredados con protección. No pueden acceder a los miembros protegidos de la clase base a través de referencias a ésta.

Constructores:

Si la clase derivada no hace una llamada explícita a un constructor de la clase base, el compilador de C# usará implícitamente un constructor de la forma :base().

* Una clase sin clases base explícitas extiende implícitamente la clase System.Object, que contiene un constructor público sin parámetros (por defecto).
* Si una clase no contiene ningún constructor, el compilador utilizará inmediatamente el constructor por “defecto”.
* El compilador no creará un constructor por defecto si una clase tiene su propio constructor explícito.
* No obstante, el compilador generará un mensaje de error si el constructor indicado no coincide con ningún constructor de la clase base.

Clases Selladas:

* La mayor parte de las clases son autónomas y **no están diseñadas** para que otras clases deriven de ellas.
* Para que el programador pueda comunicar mejor sus intenciones al compilador y a otros programadores, C# permite declarar una clase como sealed (sellada).
* La derivación de una clase sellada no está permitida (no se puede heredar de ella).

**Abstracción**

Consiste en seleccionar las características mas relevantes y comportamientos en común dentro de un conjunto de objetos, definiendo nuevos tipos de entidades.

Clases Abstractas

* No se puede crear una instancia de una clase abstracta directamente, **y es un error en tiempo de compilación utilizar el operador new en una clase abstracta**.
* Aunque es posible tener variables y valores cuyos tipos en tiempo de compilación sean abstractos, tales **variables y valores serán null** o contendrán referencias a instancias de clases no abstractas derivadas de los tipos abstractos.
* Se permite que una clase abstracta contenga miembros abstractos, aunque no es necesario.
* **No se puede sellar una clase abstracta.**
* Cuando una clase no abstracta se deriva de una clase abstracta, la clase no abstracta debe incluir implementaciones reales de todos los miembros abstractos heredados; por lo tanto, reemplaza a estos miembros abstractos.
* **Las clases abstractas se sitúan en la cima de la jerarquía de clases.**
* Establecen la estructura y significado del código.
* Facilitan la creación de marcos de trabajo, esto es posible ya que las clases abstractas poseen una información y un comportamiento común a todas las clases derivadas de un marco de trabajo.

Miembros Abstractos

Dada la definición formal, podríamos decir:

* Una declaración de método abstracto introduce un nuevo método virtual, pero no proporciona una implementación del método.
* Por esto es necesario que las clases derivadas no abstractas proporcionen su propia implementación mediante el reemplazo del método.
* Debido a que un método abstracto no proporciona una implementación real, el cuerpo-del-método de un método abstracto consiste simplemente en un punto y coma.
* Son métodos y propiedades que se declaran sin implementación.

Métodos Virtuales:

* El modificador virtual sirve para métodos, propiedades, indexadores o evento declarado y permitir que este sea anulado en una clase derivada.
* Cuando una declaración de método de instancia incluye un modificador virtual, se dice que este es un método virtual.
* Si no existe un modificador virtual, se dice que el método es un método no virtual.
* La implementación de un método no virtual es invariable, o sea 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, en 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.**
* En la invocación de un método virtual, el tipo en tiempo de ejecución de la instancia para la que tiene lugar la invocación determina la implementación del método real a invocar.
* Cuando se invoca un método no virtual, el factor determinante es el tipo en tiempo de compilación de la instancia.

Redefinición de Métodos:

* El modificador override es necesario para ampliar o modificar la implementación abstracta o virtual de un método, propiedad, indixador o evento heredado.
* Un método override proporciona una nueva implementación de un miembro que se hereda de una clase base. El método invalidado por una declaración override se conoce como método base invalidado.
* El método base reemplazado debe tener la misma firma que el método override.
* **No se puede reemplazar un método estático o no virtual. El método base reemplazado debe ser virtual, abstract u override.**
* **Una declaración override no puede cambiar la accesibilidad** del método virtual. El método override y el método virtual deben tener el mismo modificador de nivel de acceso.
* **No** se pueden usar los modificadores **new**, static o virtual para modificar un método override.
* Una declaración de propiedad de invalidación debe especificar exactamente el mismo modificador de acceso, tipo y nombre que la propiedad heredada, y la propiedad invalidada debe ser virtual, abstract u override.

**Polimorfismo**

Es la propiedad que tienen los objetos de permitir invocar genéricamente un comportamiento (método) cuya implementación será delegada al objeto correspondiente recién en tiempo de ejecución. En otras palabras, es la capacidad de tratar objetos diferentes de la misma forma.

El polimorfismo basado en herencia implica la definición de métodos en una clase base y sobrescribirlos con nuevas implementaciones en clases derivadas.

* La definición del método reside en la clase base.
* Se utiliza virtual.
* La implementación del método reside en la clase derivada.
* Se utiliza override.
* La invocación es resuelta en tiempo de ejecución.

**Clases Estáticas**

* Las clases estaticas NO heredan
* ni son heredadas
* las clases estaticas NO hacen nada
* no heredan, no pueden ser heradadas
* no pueden ser instanciadas