* **Variables estáticas o de clase**

Son propias únicamente de la clase y no de los objetos que pueden crearse de la misma, por lo tanto, sus valores son compartidos por todos los objetos de la clase. Van precedidas del modificador static.

Para invocar a una variable estática no se necesita crear un objeto de la clase en la que se define:

* Si se invoca desde la clase en la que se encuentra definido, basta con escribir su nombre.
* Si se le invoca desde una clase distinta, debe anteponerse a su nombre, el de la clase en la que se encuentra seguido del operador punto (.) <NombreClase>.variableEstatica

Suelen emplearse para definir constantes comunes a todos los objetos de la clase.

* **Ciclos de vida de las variables**

• Variables de instancia (u objeto):

– Se crean cuando se crea el objeto que las contiene.

– Se inicializan por defecto si no se hace de modo explícito:

• 0 para números, "false" para booleano, "null" para objetos.

– Se destruyen cuando el recolector de basura de Java no encuentra referencias activas para el objeto.

• Variables estáticas (o de clase):

– Se crean cuando la clase se usa por primera vez.

– Se inicializan por defecto si no se hace de modo explícito:

• 0 para números, "false" para booleano, "null" para objetos

– Suelen existir para el resto del programa (salvo que no esté cargado).

• Variables locales (o de bloque):

– Creadas en la sentencia en la que están definidas.

– No se inicializan por defecto. Contienen datos imprevisibles.

– Se destruyen al salir del bloque (en la llave final).

* **Memoria dinámica**

Es memoria que se reserva en tiempo de ejecución. Su principal ventaja frente a la estática, es que su tamaño puede variar durante la ejecución del programa. El uso de memoria dinámica es necesario cuando no conocemos el número de datos/elementos a tratar; sin embargo es algo más lento, ya que el tiempo ejecución depende del espacio que se va a usar.

Hay que mencionar que la memoria estática es más rápida ya que está disponible desde que se inició el programa.

Su tamaño y forma es variable a lo largo de un programa, por lo que se crean y destruyen en tiempo de ejecución. Esto permite dimensionar la estructura de datos de una forma precisa: se va asignando memoria en tiempo de ejecución según se va necesitando

* **Clase**

Una clase es una construcción que permite crear tipos personalizados propios mediante la agrupación de variables de otros tipos, métodos y eventos.

Una clase es como un plano. Define los datos y el comportamiento de un tipo. Si la clase no se declara como estática, el código de cliente puede utilizarla mediante la creación de objetos o instancias que se asignan a una variable. La variable permanece en memoria hasta que todas las referencias a ella están fuera del ámbito. En ese momento, CLR la marca como apta para la recolección de elementos no utilizados.

 Si la clase se declara como estática, solo existe una copia en memoria y el código de cliente solo puede tener acceso a ella a través de la propia clase y no de una variable de instancia.

* **Objeto**

 Un objeto es básicamente un bloque de memoria que se ha asignado y configurado en función del plano. Un programa puede crear muchos objetos de la misma clase. Los objetos también se denominan instancias y se pueden almacenar en una variable con nombre o en una matriz o colección. El código cliente es el código que utiliza estas variables para llamar a los métodos y obtener acceso a las propiedades públicas del objeto. En un lenguaje orientado a objetos como C#, un programa típico se compone de varios objetos que interactúan dinámicamente.

* **Instancia**

Una clase es como la definición de un objeto, pero no es el objeto en sí, del modo como una idea no es una cosa física (una silla). Así que para sentarnos necesitaremos convertir esa idea en algo, en un objeto real; a ese objeto lo llamamos instancia.

En un mismo proyecto puedo tener una o más instancias de una misma clase sin problemas.

Cada vez que creamos una nueva instancia, ésta adquiere las propiedades, métodos y eventos de la clase a la que pertenece (es lo que permite la relación es un), sin embargo, cada instancia es independiente de las otras.

* **Herencia**

La herencia es la transmisión del código entre unas clases y otras. Para soportar un mecanismo de herencia tenemos dos clases: la clase padre y la/s clase/s hija/s. La clase padre es la que transmite su código a las clases hijas. En muchos lenguajes de programación se declara la herencia con la palabra "extends".

Class Hija extends Padre { }

Eso quiere decir que todo el código de la clase padre se transmite, tal cual, a la clase hija. Si lo quieres ver así, es como si tuvieras escrito, línea a línea, todo el código de la class "Padre" dentro de las llaves de la class "Hija".

Por eso, la herencia es fundamental para reutilizar código, porque no necesitas volver a incorporar el código de Padre en Hija, sino que realmente al hacer el "extends" es como si ya estuviera ahí.

* **Sobrecarga**

Sobrecarga de funciones (Programación). Es un mecanismo que permite asignar el mismo nombre a funciones distintas. Para el compilador estas funciones no tienen nada en común a excepción del identificador, por lo que se trata en realidad de un recurso semántico del lenguaje que solo tiene sentido cuando se asigna el mismo nombre a funciones que realizan tareas similares en objetos diferentes.

Al tratar de los operadores, para el compilador son en realidad funciones, cuyos nombres y sintaxis de invocación son un tanto especiales y que la sobrecarga de estas funciones permite, por ejemplo, extender los conceptos de suma (+), asignación (=) o identidad (==), a objetos distintos de los básicos (preconstruidos en el lenguaje). El hecho de que acciones distintas pero conceptualmente semejantes, puedan ser representadas por el mismo operador (función), resulta a la postre una gran ayuda conceptual. Por ejemplo: puede hablarse de una "suma" de enteros y de una "suma" de complejos que serán gobernadas por versiones distintas del operador suma (+).

Lenguajes y Paradigmas de la Prgramacion

* **Ensombrecimiento**

El problema de “ensombrecimiento” se produce cuando en un ámbito de validez se define una variable con nombre idéntico a otra válida en un ámbito superior.

Se llama shadowing al hecho de que en una clase una variable miembro y una variable local definida en un método miembro, se llamen igual.

* **Ciclo de vida de variable de bloque**

• Variables locales (en un método o bloque)

– Existen desde el punto de definición hasta el final del bloque

• Los bloques se definen mediante llaves { }

• Los bloques se suelen utilizar para definir:

– El cuerpo del método

– Sentencias múltiples en operaciones de tipo if-else o en forma de bucles

– Las variables locales sólo existen dentro del propio método

• No se da acceso a ningún otro método

– Si la variable local y de instancia tienen el mismo nombre, la variable local oculta a la de instancia: