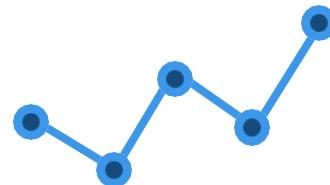
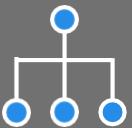


LENGUAJES DE PROGRAMACIÓN II

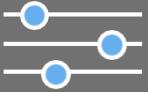




BIENVENIDA

Bienvenido(a) a la asignatura *Lenguajes de Programación II* En el curso anterior analizamos e implementamos conceptos como clase y objeto. En este curso nos adentraremos más en la programación orientada a objetos, en particular a conceptos como herencia, clase base y derivada, así como métodos, sobrecarga y encapsulamiento. Aunque utilizaremos los conceptos del lenguaje de C++, estos pueden implementarse en otros lenguajes de programación.





LIBROS RECOMENDADOS

- Deitel, P. & Deitel, H. (2017). *C++ How to Program*. Pearson.
- Nesteruk, D. (2018). *Design Patterns in Modern C++: Reusable Approaches for Object-Oriented Software Design*. Apress.



1

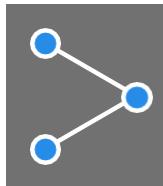
UNIDAD

HERENCIA

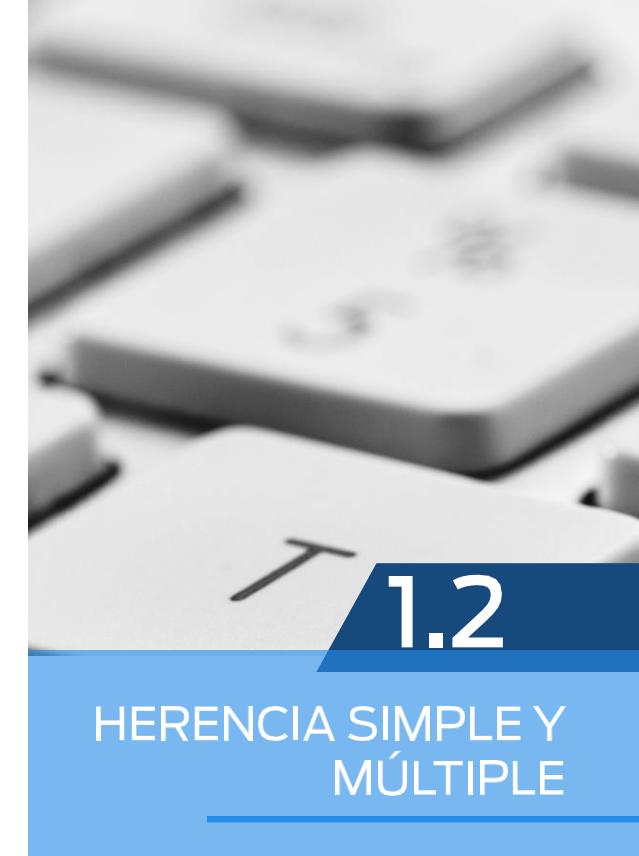
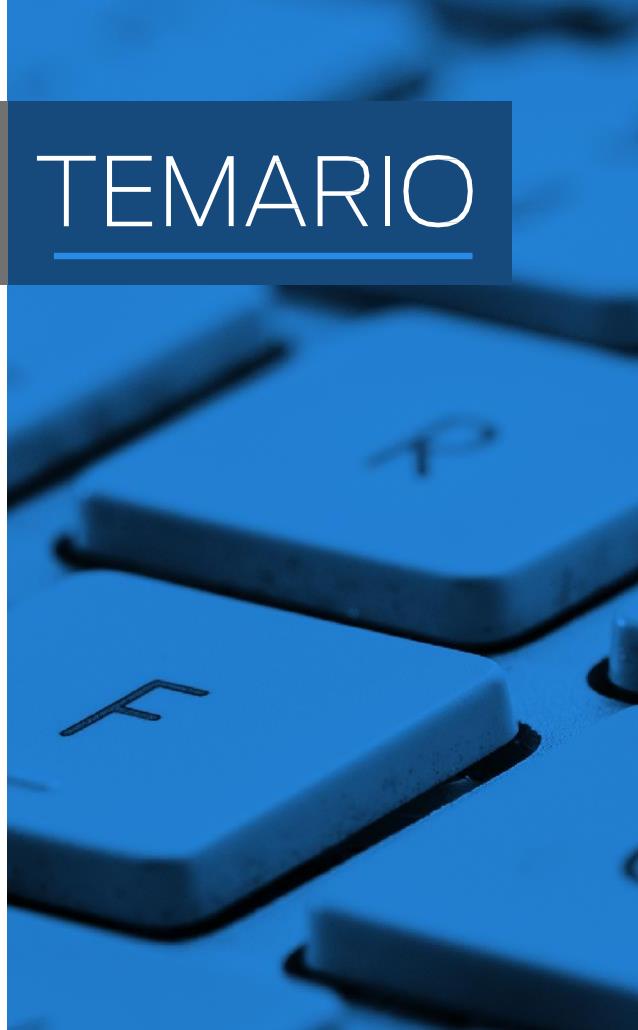


1.1

CLASE BASE Y
DERIVADA



TEMARIO



1.2

HERENCIA SIMPLE Y
MÚLTIPLE

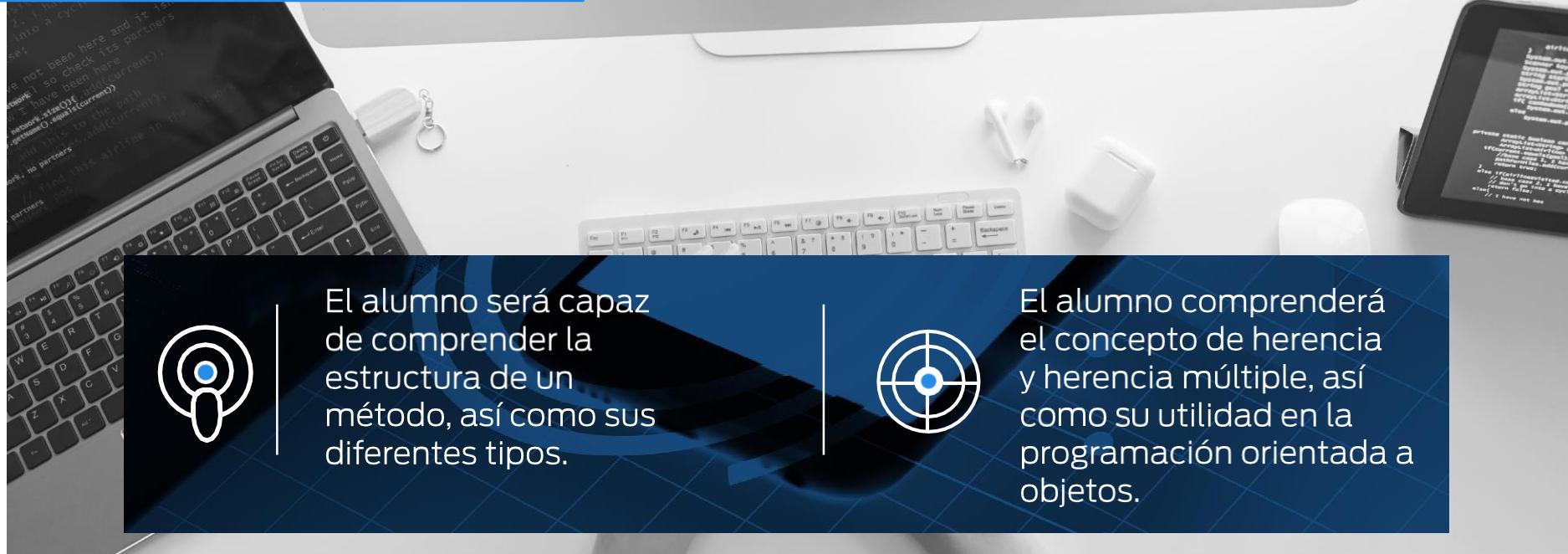


INTRODUCCIÓN

La asignatura *Lenguajes de Programación II* tiene como objetivo analizar el paradigma de programación que usa objetos. De esta manera, lograr el diseño de aplicaciones y programas orientados a objetos (C++).

En esta primera unidad aprenderás sobre la herencia. Este concepto es básico y permite poder crear estructuras abstractas y jerárquicas que ayudan a modelar los sistemas de información como si fuesen una réplica del mundo real.

COMPETENCIAS A DESARROLLAR



El alumno será capaz de comprender la estructura de un método, así como sus diferentes tipos.



El alumno comprenderá el concepto de herencia y herencia múltiple, así como su utilidad en la programación orientada a objetos.

VIDEO



Te invitamos a ver el siguiente video:



CLASE BASE Y DERIVADA

La **herencia** es el proceso mediante el cual un objeto de una clase adquiere propiedades definidas en otra clase que la precede en una jerarquía de clasificaciones. Además, permite la definición de un nuevo objeto a partir de otros, al agregar diferencias entre ellos (programación diferencial).

Las clases heredan los datos y métodos de una súper clase. Un método heredado puede ser sustituido por uno propio, si es que ambos tienen el mismo nombre.



CLASE BASE Y DERIVADA

La herencia es la propiedad que permite a los objetos construirse a partir de otros objetos

La utilidad de la herencia radica en que las clases hijas reciban propiedades de sus clases padres, evitando la necesidad de repetir el código anterior. Esto permite hacer el código reutilizable.



CLASE BASE Y DERIVADA

Ventaja:

- Permite reutilizar código, extendiendo su funcionalidad.

Desventajas:

- Se introduce una fuerte dependencia en la clase hija respecto de la clase padre.
- Puede dificultar la reutilización.
- Un cambio en la clase padre puede tener efectos imprevistos en la clase hija.



CLASE BASE Y DERIVADA

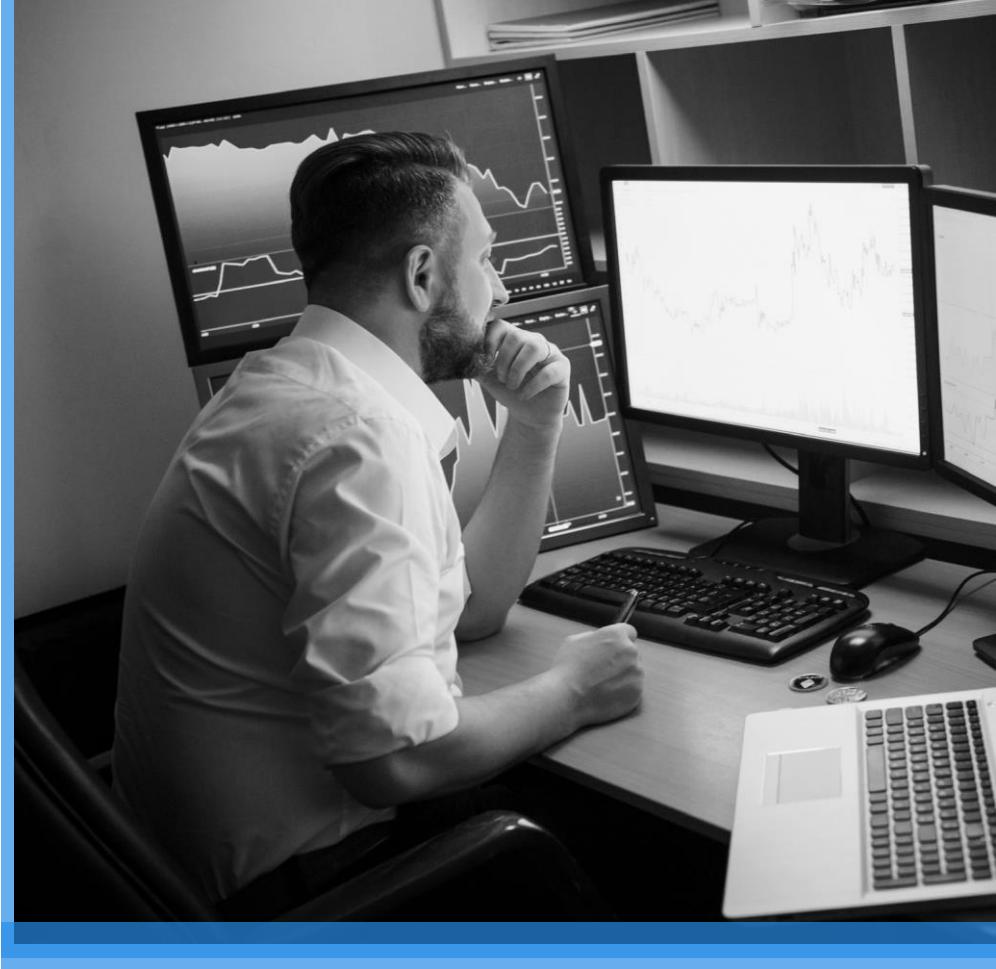
- Un objeto de una clase hija puede tener un comportamiento inconsistente con lo esperado de un objeto de la clase padre
- Se establece una jerarquía o clasificación.
- Si cambia el criterio de clasificación puede acarrear muchas modificaciones.

Por lo anterior, es importante diferenciar las propiedades de una clase base y una derivada.

```
<div style="background-image: url('img/todos.png'); background-size: cover; height: 100px; width: 100%;>
  <p>The image can be tiled across the top</p>
  // persisted properties
  <html> <p style="font-weight:bold;">HTML font code is now using</p>
  <html> <body style="background-color:yellowgreen; color: black; text-align: center; text - :200px;"> <.todolistid = data.todoitem.todosId ->
    <html> <errorMessage = ko , observable() : = ko , observable()>persisted properties</html>
    <p style="color:orange;">HTML font code is now using</p>
    function todoitem(data) {
      var self = this <html> <errorMessage = ko , observable()>
      data = dta || <html> <errorMessage = ko , observable()>
    }
  // Non - persisted properties
  <html> <errorMessage = text - :200px;>ko , observable()>
  <p style="font-weight:bold;">HTML font code is now using</p>
  <body style="background-color:yellowgreen; color: black; text-align: center; text - :200px;"> <.todolistid = data.todoitem.todosId ->
    <html> <errorMessage = ko , observable() : = ko , observable()>persisted properties</html>
    <errorMessage = ko , observable() : = ko , observable()>
    <p style="color:orange;">HTML font code is now using</p>
```

CLASE BASE Y DERIVADA

- **Clase base.** Es aquella clase en la que ninguno de sus atributos u objetos depende de alguno otro. En términos de herencia, sería la clase padre; la que siempre se mantiene fija.
- **Clase derivada.** Es aquella que depende de la clase base. Algunos de sus métodos son también heredados. Muchas veces, el compilador arrojará malos resultados, ya que, al ser dependiente esta clase, se pudieran generar errores lógicos



CLASE BASE Y DERIVADA

Entonces, una clase derivada es la que se define en función de otra clase. Su sintaxis es muy sencilla. Se declara la clase como siempre, pero después de nombrarla escribimos dos puntos (:) y el nombre de su clase base. Esto indica al compilador que todos los miembros de la clase base se heredan en la nueva. La forma general de la declaración de clases derivadas es la siguiente:

```
class <clase_derivada> :  
[public|private] <base1> [, [public|private] <base2>] {};
```

Por ejemplo, si tenemos la clase *empleado* (derivada de *persona*), y queremos definir la clase *directivo*, entonces podemos declarar esta última como derivada de la primera.

CLASE BASE Y DERIVADA

Así, un *directivo* tendrá las características de *persona* y de *empleado*, pero definirá, además, unos nuevos atributos y métodos propios de su clase:

```
class directivo : empleado {  
private:  
    long num_empleados;  
    long num_acciones;  
  
    ...  
public:  
    ...  
    void despide_a (empleado *e);  
    void reunión_con (directivo *d);  
    ...  
};
```

CLASE BASE Y DERIVADA

En lugar de derivar una clase de la base que reúna las características, podemos derivar una clase de las subclases que las incorporen.

Por ejemplo, si definimos una clase *ventana* y derivamos las clases *ventana_con_borde* y *ventana_con_menu*, en lugar de derivar de la clase *ventana* una clase *ventana_con_menu_y_borde*, la derivamos de las dos subclases.

A continuación, un ejemplo:



CLASE BASE Y DERIVADA

```
// Clase base Persona:  
class Persona {  
public:  
    Persona(char *n, int e);  
    const char *LeerNombre(char  
    *n) const;  
    int LeerEdad() const;  
    void CambiarNombre(const char  
    *n);  
    void CambiarEdad(int e);  
protected: char nombre [40];  
int edad;  
};
```

```
// Clase derivada Empleado:  
class Empleado : public  
Persona {  
public: Empleado(char *n, int  
e, float s);  
float LeerSalario() const;  
void CambiarSalario(const  
float s);  
protected: float  
salarioAnual;  
};
```

CLASE BASE Y DERIVADA

Cuando una clase hereda otra, la clase derivada incorpora todos los miembros de la clase base además de los propios.

La herencia es una herramienta importante en el desarrollo de aplicaciones. Principalmente para:

- Organizar el diseño.
- Reusar las clases (propias o no).
- Mejorar el mantenimiento.



CLASE BASE Y DERIVADA

La derivación de clases tiene que ver con la creación de una clase a la que se denomina clase base. En ella se definen un conjunto de propiedades (atributos y métodos) que se considerarán como los más generales.

Una vez que se tenga la certeza de haber dado con la clase general, podemos crear otra clase que, en cierto modo, supone una especialización de la anterior.



CLASE BASE Y DERIVADA

Dado que la clase derivada es una especialización de la anterior, además de las propiedades definidas para la clase base, tendrá las propias. Estas servirán para diferenciarla de la anterior

Para lograr que se consiga esa especialización, solo hace falta heredar de la clase base los atributos que nos interesan. Lo único nuevo que se tiene que escribir serán los atributos que especializan esa clase. A la clase que hereda de otra se le llama clase derivada.



```
data-val="popularity" data-track="click.searchFilters.sort-popular">
> <span class="hidden-xs hidden-sm">Meest relevant</span>
> <span class="label-selectable-pill hidden-md hidden-lg active">
</span>
</li>
<li class="js_close-drawer">
<button title="Nieuwe inhoud" class="btn btn-link navbar-btn js_search-filter-sort" data-val="newest" data-track="click.searchFilters.sort-recent">
> <span class="hidden-xs hidden-sm">Nieuwe inhoud</span>
> <span class="label-selectable-pill hidden-md hidden-lg active">
</span>
</li>
<input type="hidden" name="sort" value="popular"/>
</ul>
</li>
<div class="drawer js_drawer">
<ul class="nav navbar-nav">
<li class="js_close-drawer close-drawer hidden-md hidden-lg">
<li><button class="btn btn-link navbar-btn dropdown-toggle" data-val="filter" data-track="click.navbar-filter">
<li class="filter-horizontal">
<ul class="filter-content nav buttonize navbar-nav" data-val="filter-content" data-track="click.navbar-filter-content">
<li class="js_close-drawer">
<span class="label-selectable-pill hidden-md hidden-lg active">
</span>
</li>
</ul>
</li>
</ul>
</div>
```



CLASE BASE Y DERIVADA

Una clase derivada hereda todos los miembros de datos y métodos de la clase base. Sin embargo, los métodos propios de la clase derivada no tendrán acceso a los miembros privados de la clase base. Los miembros públicos de una clase base serán también públicos en cualquier clase derivada de esta.

Además, pueden agregarse nuevos miembros (datos y métodos) a la clase derivada. Estos no formarán parte de la clase base.

CLASE BASE Y DERIVADA

Cuando utilizamos la herencia al definir una clase, podemos decir que un objeto de la clase derivada sea, al mismo tiempo, un objeto de la clase base. Por ejemplo:

```
Vehiculo v(900); //Vehículo con peso 900  
Coche a(1200,130,"Ford"); //peso=1200, vel=130 y marca Ford
```

Como *Coche* es, a su vez, un *Vehículo*, podríamos hacer la asignación:

```
v=a; //El objeto v guardará el valor 1200 en su atributo peso
```

También, podemos intentar:

```
a=v; //El objeto a recibiría el peso, pero no la velocidad ni el nombre.  
//Esta asignación NO será aceptada por el compilador.
```

CLASE BASE Y DERIVADA

```
import java.util.ArrayList; import java.util.Scanner; import java.io.IOException; import java.util.Arrays; import java.util.List;
```

```
public class AirlineProblem {
```

```
    public static void main(String[] args){
```

```
        Scanner scannerToReadAirlines = null; if (main(String[] args)){
```

```
            try{
```

```
                scannerToReadAirlines = new Scanner(new File("airlines.txt")); if (null);
```

```
            } catch (IOException e){
```

```
                System.out.println("could not connect to file airlines.txt"); Scanner(new File("airlines.txt")); System.exit(0);
```

```
            }
```

```
            if(scannerToReadAirlines != null){Exception e){
```

```
                ArrayList<Airlne> airlinesPartnersNetwork = new ArrayList<Airlne>();
```

```
                Airlne newAirlne; String lineFromfile; System.out.println("could not connect to file airlines.txt");
```

```
                string[] airlineNames; string[] airlineNames;
```

```
                while(scannerToReadAirlines.hasNextLine());
```

```
                lineFromfile = scannerToReadAirlines.nextLine();
```

```
                airlineNames = lineFromfile.split(","); for (AirlnePartnersNetwork <- n
```

```
                newAirlne = new Airlne(airlineNames); line;
```

```
                airlinesPartnersNetwork.add(newAirlne);
```

```
                String lineFromfile;
```

```
                System.out.println(airlinesPartnersNetwork);
```

```
                Scanner keyboard = new Scanner(System.in);
```

```
                String inputName; if (inputName.equals("q")){
```

Dado que una clase derivada contiene todos los miembros de la clase base, una instancia de la primera se puede emplear de forma automática como si fuera una instancia de la clase base.

El caso contrario no es cierto, no se puede tratar un objeto de la clase base como si fuera un objeto de una clase derivada, ya que el objeto de la clase base no tiene todos los miembros de la clase derivada.

VIDEO



Te invitamos a ver el siguiente video:



HERENCIA SIMPLE Y MÚLTIPLE

La característica más distingible del paradigma orientado a objetos es la herencia.

- **Herencia simple.** Cuando un objeto hereda una característica.
- **Herencia múltiple.** Cuando un objeto hereda características de más de un objeto principal.





HERENCIA SIMPLE Y MÚLTIPLE

Por ejemplo, las clases *Docente* y *Estudiante* heredan las propiedades de la clase *Persona* (superclase, herencia simple). La clase *Preparador* (subclase) hereda propiedades de la clase *Docente* y de la clase *Estudiante* (herencia múltiple).

HERENCIA SIMPLE Y MÚLTIPLE

Un ejemplo de la herencia múltiple puede ser con una analogía con los autos anfibios. Estos pueden circular por tierra y mar. En este sentido, podemos definir a los anfibios como elementos que heredan características de los vehículos terrestres y los marinos.

En la sintaxis, primero se pone el nombre de la nueva clase, después dos puntos (:) y, finalmente, la lista de clases padre:

```
class anfibio : terrestre, marino {  
    ...  
};
```

HERENCIA SIMPLE Y MÚLTIPLE

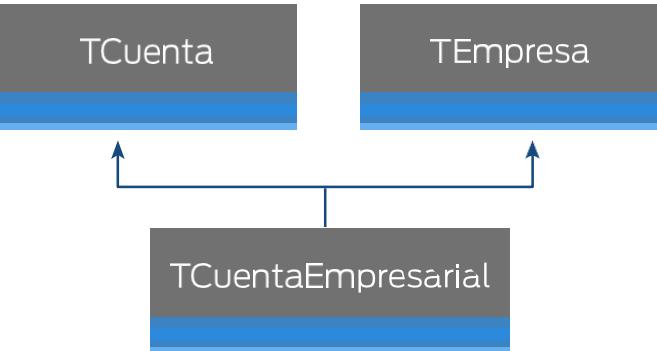
La herencia múltiple es el mecanismo que permite al programador hacer clase derivadas a partir, no solo de una clase, sino de varias. Ejemplo:

```
class Tempresa{  
protected:  
string nomEmpresa;  
public:  
Tempresa(string unaEmpresa) : nomEmpresa(unaEmpresa)  
{};  
void setNombre(string nuevoNombre){  
nomEmpresa = numvoNombre;}  
-Tempresa(){}  
};
```

HERENCIA SIMPLE Y MÚLTIPLE

Continuación:

```
class Tempresa{  
protected:  
string nomEmpresa;  
public:  
Tempresa(string unaEmpresa) :  
nomEmpresa(unaEmpresa)  
{};  
void setNombre(string nuevoNombre){  
nomEmpresa = numvoNombre;}  
-Tempresa(){}  
};
```





HERENCIA SIMPLE Y MÚLTIPLE

La herencia en C++ es un mecanismo de abstracción creado para poder facilitar y mejorar el diseño de las clases de un programa. Con ella se pueden crear nuevas clases a partir de clases ya establecidas, siempre y cuando exista un tipo de relación especial.

En la herencia, las clases derivadas *heredan* los datos y funciones miembro de las clases base; sin embargo, la parte privada de una clase no es directamente accesible desde la clase derivada.

HERENCIA SIMPLE Y MÚLTIPLE

Los constructores no se heredan:

- Siempre son definidos por las clases derivadas
- Para la creación de un objeto de clase derivada se invoca a todos los constructores de la jerarquía
- De acuerdo al orden de ejecución de los constructores, primero se ejecuta el constructor de la clase base y luego el de la derivada.





HERENCIA SIMPLE Y MÚLTIPLE

El destructor no se hereda:

- Siempre es definido por la clase derivada.
- Para la destrucción de un objeto de clase derivada se invoca a todos los constructores de la jerarquía
- Primero se destruye el objeto derivado y luego el objeto base.
- Se hace una llamada implícita al destructor de la clase base.

HERENCIA SIMPLE Y MÚLTIPLE

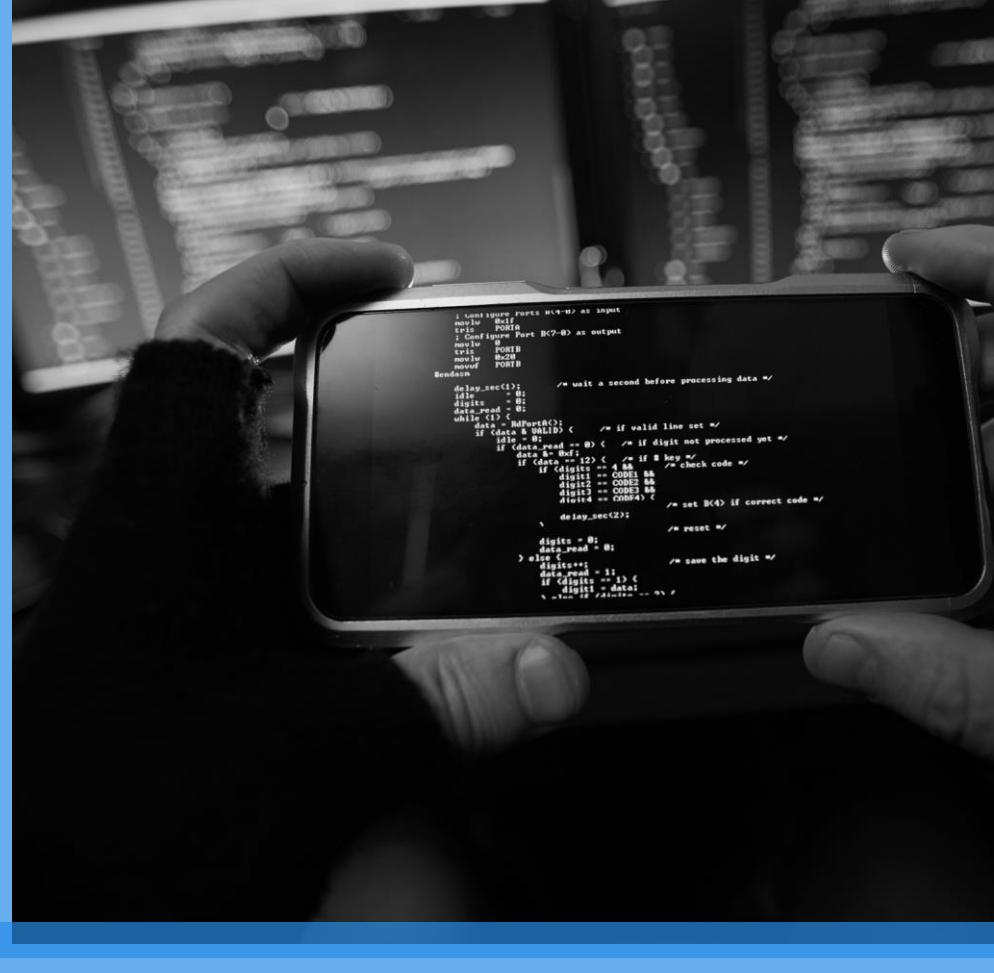
Además de los constructores y el destructor, los siguientes elementos de la clase **no se heredan:**

- Funciones amigas
- Funciones estáticas de la clase
- Datos estáticos de la clase
- Operador de asignación sobrecargado
- Las clases base



HERENCIA SIMPLE Y MÚLTIPLE

En C++ es posible definir clases abstractas. Esta es una clase que está diseñada solo como clase padre, de las cuales se deben derivar clases hijas. Se utiliza principalmente para representar aquellas entidades o métodos que después se implementarán en las clases derivadas. En sí, la clase abstracta no contiene ninguna implementación, solamente representa los métodos que se deben implementar.





HERENCIA SIMPLE Y MÚLTIPLE

La clase abstracta contiene una o más funciones virtuales puras. Si una clase derivada no redefine una función virtual pura, la clase derivada la hereda como función virtual pura y se convierte también en clase abstracta.

Por el contrario, aquellas clases derivadas que redefinen todas las funciones virtuales puras de sus clases base reciben el nombre de clases derivadas concretas.

HERENCIA SIMPLE Y MÚLTIPLE

Las clases abstractas actúan como expresiones de conceptos generales de los que pueden derivarse más clases concretas. No se puede crear un objeto de un tipo de clase abstracta.

Se le considera así, si contiene al menos una función pura virtual. Las clases derivadas de esta deben implementar la función virtual pura o deben ser también clases abstractas.





HERENCIA SIMPLE Y MÚLTIPLE

No se pueden usar clases abstractas para:

- Variables o datos miembro.
- Tipos de argumento
- Tipos de valor devuelto de función.
- Tipos de conversiones explícitas.

Si el constructor para una clase abstracta llama a una función virtual pura, directa o indirectamente, el resultado queda sin definir. Sin embargo, los constructores y el destructor pueden llamar a otras funciones miembro.

VIDEO



Te invitamos a ver el siguiente video:



FORO 1

Entorno de trabajo.

Participa en el foro enviando imágenes que demuestren que ya tienes acceso a las siguientes herramientas en su versión de prueba:

- Eclipse
- NetBeans
- Visual Studio

Presiona el botón para participar en el foro.



La herencia en C++ es un mecanismo de abstracción creado para poder facilitar y mejorar el diseño de las clases de un programa. Con ella se pueden crear nuevas clases a partir de clases ya establecidas, siempre y cuando exista un tipo de relación especial.

En la herencia, las clases derivadas heredan los datos y funciones miembro de las clases base.

La parte privada de una clase no es directamente accesible desde la clase derivada.

CONCLUSIÓN



```
(function (ko, datacontext) {}  
  <div style="background-image:url(1px solid black); background-color:#f0f0f0; height: 200px;">  
    <p>The image can be tiled across the background relatively easily.  
  </div>  
  
  || persisted properties  
  
<html> <p style="font-weight:bold;">HTML font weight  
<html> <body style="background-color:yellow; width:100%; height:100%;">  
<html> <.todolistid = data.todoId;  
  
  || Non - persisted properties  
<html> <errorMessage = ko.observable().
```

¡FELICIDADES!

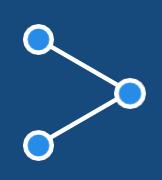
Acabas de concluir la [primera unidad](#) de tu curso *Lenguajes de Programación II*. Te invitamos a finalizar este esfuerzo realizando el examen parcial correspondiente. Para ello, debes regresar a la pantalla principal y dar clic en *Presentar examen*.

2

UNIDAD

MÉTODOS





2.1

DECLARACIÓN Y
ATRIBUTOS



TEMARIO



2.2

ARGUMENTOS Y
PARÁMETROS





INTRODUCCIÓN

En esta segunda unidad aprenderás sobre los métodos. Antes que nada, hay que saber que un objeto es una abstracción en la que se unen sentencias y datos. En este sentido, este solo podrá ser tratado con sus métodos predefinidos.

En programación estructurada se trabaja con funciones. Sin embargo, en la programación orientada a objetos este concepto cambia por métodos, los cuales reflejan el comportamiento de los objetos.

COMPETENCIAS A DESARROLLAR



El alumno será capaz de comprender la estructura de un método.



El alumno será capaz de implementar métodos con distintos argumentos.

VIDEO

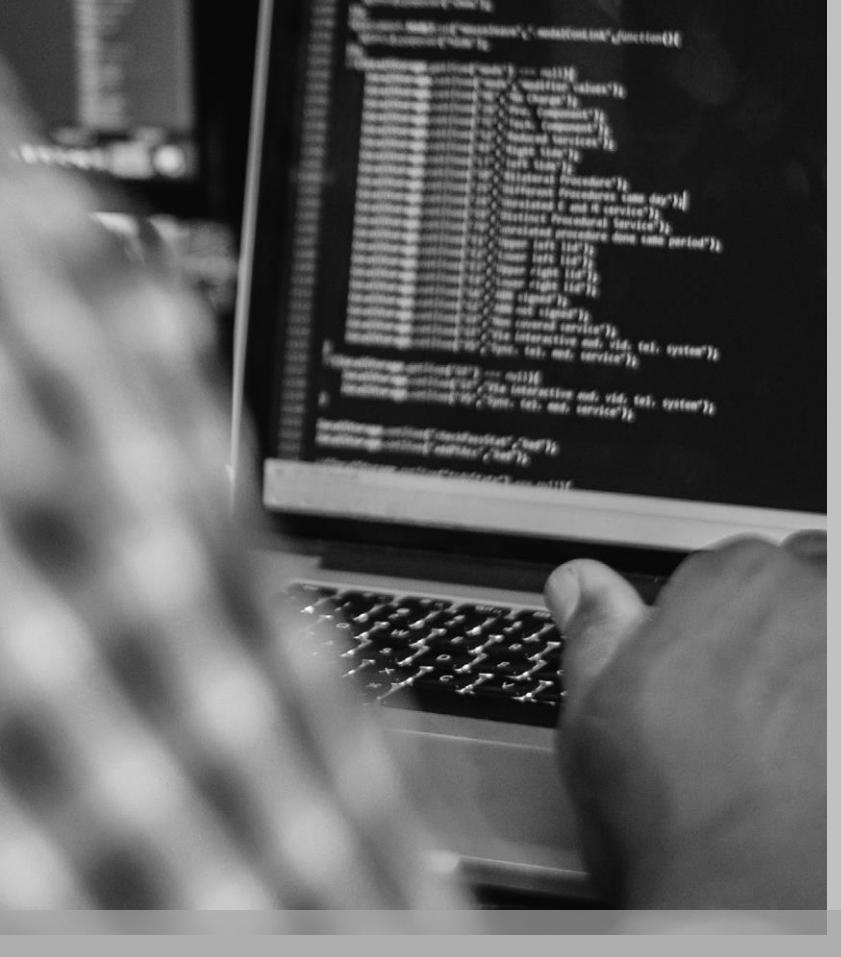


Te invitamos a ver el siguiente video:



DECLARACIÓN Y ATRIBUTOS

Las clases definen atributos y métodos. Los métodos de una clase sirven para manipular los atributos de la clase. Un método puede recibir valores, efectuar operaciones con estos y retornarlos después del procesamiento.



DECLARACIÓN Y ATRIBUTOS

Los métodos siempre presentan paréntesis al final de su nombre. Los paréntesis contienen los valores necesarios para la ejecución del método. Estos valores se llaman **argumentos**.

Ejemplo de sintaxis:

```
objeto.argumento(ArgumentosDelMétodo)
```

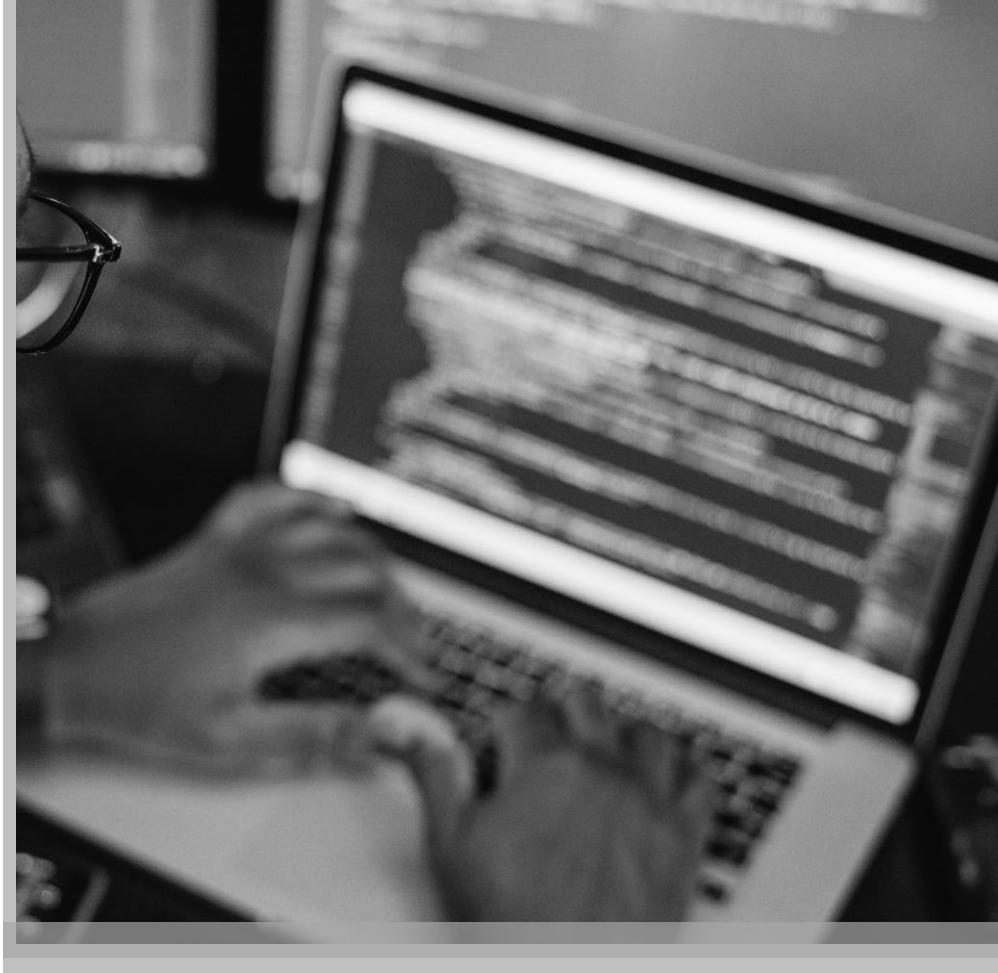
Ejemplo de implementación:

```
miCamino.gira(5); // Lo cual podría hacer que el camino  
avance a 5 km/h
```

DECLARACIÓN Y ATRIBUTOS

Cuando se llama un método se *envía un mensaje*. El control del programa pasa a ese método y lo mantendrá hasta que este finalice. La mayoría de los métodos devuelven un resultado (*return*); por ello, cuando se define un método, hay que indicar el tipo de dato al que pertenece el resultado del mismo.

Si el método no devuelve ningún resultado, se indica como tipo de datos a devolver el tipo *void* (vacío).



DECLARACIÓN Y ATRIBUTOS

Cuando una clase ya tiene definidos sus métodos, es posible invocarlos utilizando los objetos de esa clase. En esa invocación se deben indicar los parámetros (o argumentos) que cada método requiere para poder realizar su labor.

Ejemplo de uso de métodos:

```
balón.botar(); //sin argumentos  
miCoche.acelerar(10);  
ficha.comer(posición15); //posición 15 es una variable que  
pasa  
//como argumento  
partida.empezarPartida("18:15",colores);
```



DECLARACIÓN Y ATRIBUTOS

Los métodos se declaran y definen de la misma manera que una función cualquiera en una aplicación convencional de C++. Dicho de otro modo, en su firma expresan el valor de retorno, un nombre para el método y una lista de parámetros de entrada.

Usualmente se hace la declaración de los métodos al interior de la clase. Por su parte, la definición se hace por fuera.

DECLARACIÓN Y ATRIBUTOS

La sintaxis de la declaración de un método considera lo siguiente:

- **El tipo de datos o de objeto que devuelve.**

Se debe indicar si el resultado del método es un *número entero*, o *booleano* o *string*, o de una clase determinada. Si el método no devuelve valor alguno, se debe indicar como tipo de valor *void*.





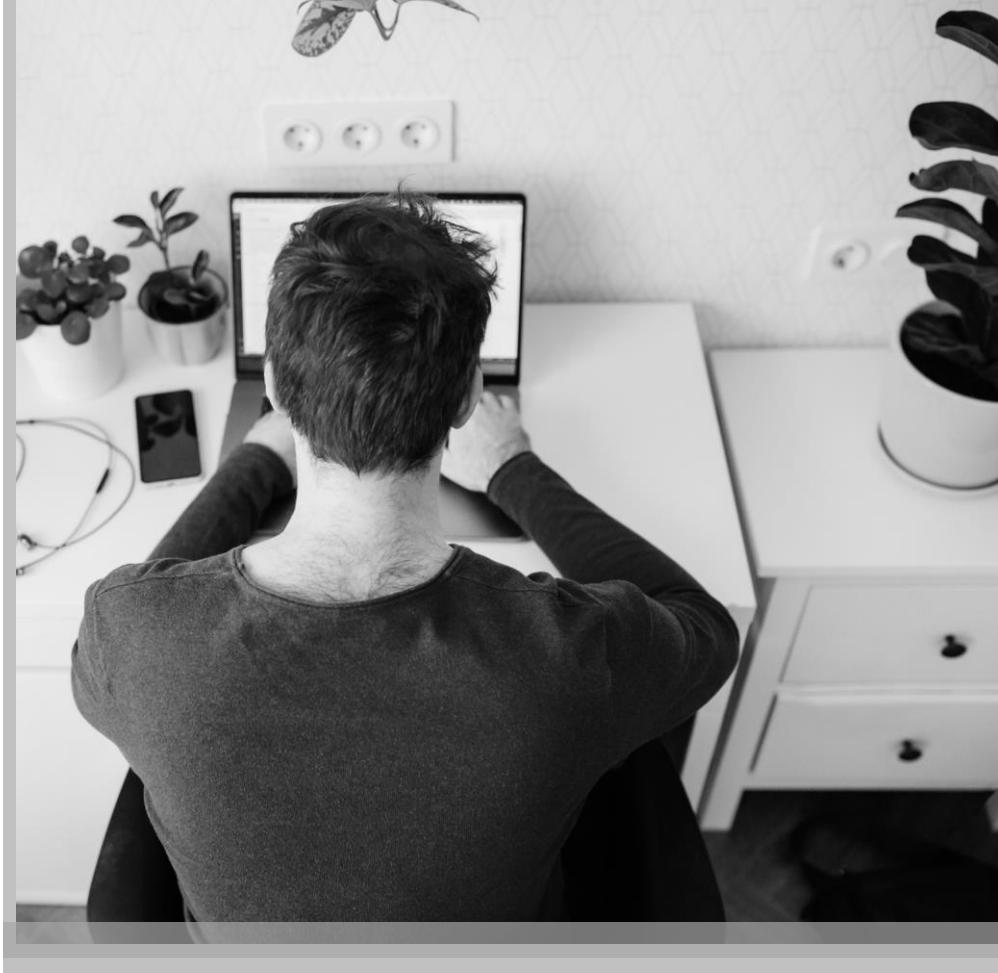
DECLARACIÓN Y ATRIBUTOS

- **El identificador del método.** Cumple las mismas reglas que los identificadores de atributos. También deben empezar con una letra minúscula.
- **Los parámetros.** Los métodos pueden necesitar datos para realizar su tarea. Estos son en realidad una lista de valores u objetos, y los tipos o clases de los mismos. Cuando el método finaliza, los parámetros se eliminan.

DECLARACIÓN Y ATRIBUTOS

- **El cuerpo del método.** Es decir, el código que permite al método realizar su tarea. Esto es lo más complicado. Dentro de este código se pueden declarar atributos, objetos y utilizar cualquier conjunto de instrucciones de C++. De la misma manera, invocar métodos de otras clases y objetos. El valor resultado del método se regresa mediante la instrucción *return*.

Estos componentes se ejemplifican a continuación:



DECLARACIÓN Y ATRIBUTOS

```
public class Vehiculo {  
    public int ruedas;  
    private double  
velocidad=0;  
    String nombre;  
    public void  
acelerar(double cantidad) {  
        velocidad += cantidad;  
    }  
    public void frenar(double  
cantidad){  
        velocidad -= cantidad;  
    }  
}
```

```
public double  
obtenerVelocidad(){  
    return velocidad;  
}  
public static void main(String  
args[]){  
    Vehiculo miCoche = new  
Vehiculo();  
    miCoche.acelerar(12);  
    miCoche.frenar(5);  
    System.out.  
println(miCoche.  
obtenerVelocidad());  
}// Da 7.0  
}
```



DECLARACIÓN Y ATRIBUTOS

En la clase anterior, los métodos *acelerar* y *frenar* son de tipo *void*. Por ello no tienen la sentencia *return*.

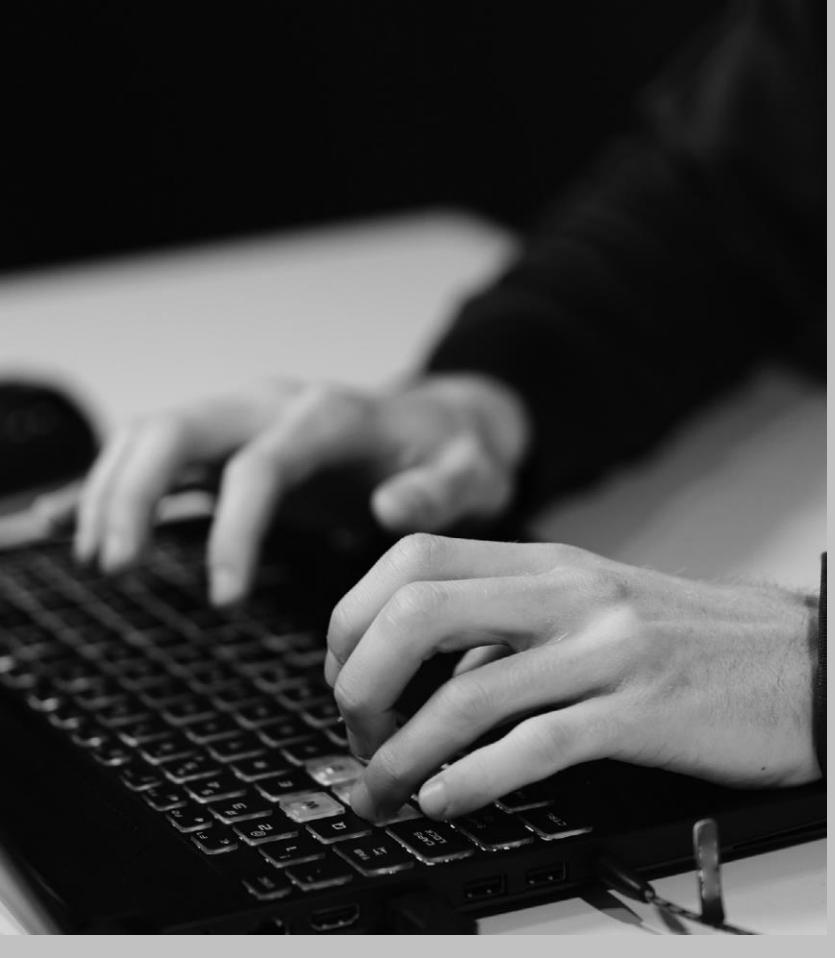
Sin embargo, el método *ObtenerVelocidad* es de tipo *double*, por lo que su resultado debe ser devuelto mediante la sentencia *return*.

Además del tipo de retorno hay otros conceptos relevantes a considerar:

DECLARACIÓN Y ATRIBUTOS

- **Alcance.** También llamado ámbito, es el rango de código en el cual el atributo es conocido. Los atributos declarados dentro de un método son locales a este. Por su parte, las que lo están en el cuerpo de la clase son miembros de esta y accesibles por todos los métodos de la clase.
- **Extensión.** También llamada vida, se refiere al tiempo de ejecución durante el cual el atributo tiene asignado un espacio de memoria para almacenar algún valor.





DECLARACIÓN Y ATRIBUTOS

Dentro de los métodos pueden incluirse:

- Declaraciones de atributos locales.
- Asignaciones a atributos.
- Operaciones matemáticas.
- Llamados a otros métodos.
- Estructuras de control.
- Excepciones.

VIDEO



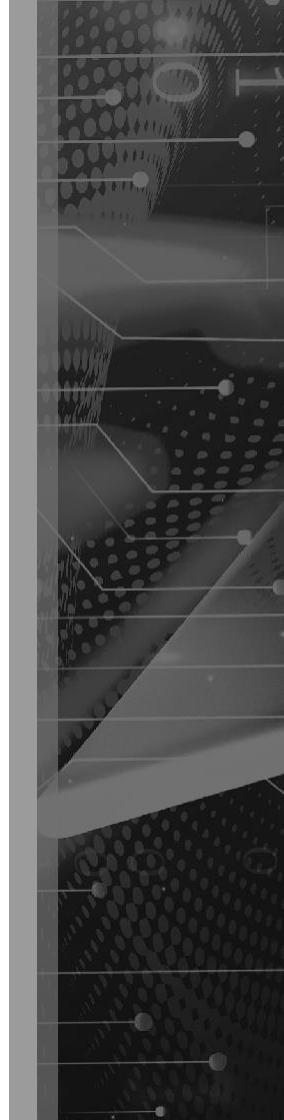
Te invitamos a ver el siguiente video:



ARGUMENTOS Y PARÁMETROS

Los argumentos son los datos que recibe un método y que necesita para funcionar.
Ejemplo:

```
public class Matemáticas {  
    public double factorial(int n){  
        double resultado;  
        for (resultado=n;n>1;n--) resultado*=n;  
        return resultado;  
    }  
    ...  
    public static void main(String args[]){  
        Matemáticas m1=new Matemáticas();  
        double x=m1.factorial(25);//Llamada al método  
    }  
}
```



ARGUMENTOS Y PARÁMETROS

En el ejemplo anterior, el valor “25” es un argumento requerido por el método factorial para que este devuelva el resultado (que será el factorial de “25”). En el código del método factorial, este valor “25” es copiado a la variable “n”, que es la encargada de almacenar y utilizar este valor.

Se dice que los argumentos son por valor, si la función recibe una copia de estos datos; es decir, la variable que se pasa como argumento no estará afectada por el código.



ARGUMENTOS Y PARÁMETROS

Ejemplo:

```
class Prueba {  
    public void metodo1(int entero){  
        entero=18;  
    ...  
}  
...  
public static void main(String args[]){  
    int x= 24;  
    Prueba miPrueba = new Prueba();  
    miPrueba.metodo1(x);  
    System.out.println(x); //Escribe 24, no 18  
}
```

ARGUMENTOS Y PARÁMETROS

El anterior ejemplo muestra el paso de parámetros por valor. La variable “x” se envía como argumento o parámetro para el método *metodo1*; ahí, la variable *entero* recibe una copia del valor de “x”, a la cual se le asigna el valor de “18”.

Cuando el código del *metodo1* finaliza, la variable *entero* desaparece. En todo el proceso, la variable “x” no ha sido afectada y seguirá valiendo “24”.



ARGUMENTOS Y PARÁMETROS

Otro ejemplo:

```
class Prueba {  
    public void metodo1(int[] entero){  
        entero [0]=18;  
        ...  
        ...  
    }  
    public static void main(String args[]){  
        int x[]={24,24};  
        Prueba miPrueba = new prueba();  
        miPrueba.metodo1(x);  
        System.out.println(x[0]); //Escribe 18, no 24  
    }  
}
```

En este podemos ver la variable “x” como array. En este caso, el parámetro *entero* recibe una referencia a “x”, no una copia.

ARGUMENTOS Y PARÁMETROS

La cuestión aquí yace en qué se pasa por referencia y qué se pasa por valor. Las **reglas** son simples:

- Los tipos básicos (*int, double, char, boolean, float, short* y *byte*) se pasan por valor.
- También se pasan por valor atributos de tipo *String*.
- Los objetos y *arrays* se pasan por referencia.



VIDEO



Te invitamos a ver el siguiente video:



ACTIVIDAD 1

Te invitamos a realizar la siguiente actividad:

Presiona el botón para descargar la actividad:



Presiona el botón para entregar la actividad:



En C++, los métodos y atributos están estrechamente relacionados por la propiedad de conjunto. Esta destaca que una clase requiere de métodos para poder tratar los atributos con los que cuenta.

El programador debe pensar indistintamente en estos conceptos, sin separar ni darle mayor importancia a ninguno de ellos. De lo contrario, se podría caer en el mal hábito de crear clases contenedoras de información, por una lado, y clases con métodos que manejen a las primeras, por el otro. De esta manera, se estaría realizando una programación estructurada camuflada en un lenguaje de programación orientado a objetos.

CONCLUSIÓN



```
(function (ko, datacontext) {}  
<div style="background-image:url('1px')"  
      background .text- todoitem;  
      height .text - :200px;">  
<p>The image can be tiled across the background, while the text  
</div>  
  
|| persisted properties  
  
<html> <p style="font-weight:bold;">HTML font code is simple.  
<html> <body style="background-color:yellow;">  
<html> <.todolistid = data.todoId,  
  
|| Non - persisted properties  
<html> <errorMessage = ko .observable()>
```

¡FELICIDADES!

Acabas de concluir la [segunda unidad](#) de tu curso *Lenguajes de Programación II*. Te invitamos a finalizar este esfuerzo realizando el examen parcial correspondiente. Para ello, debes regresar a la pantalla principal y dar clic en *Presentar examen*.

3

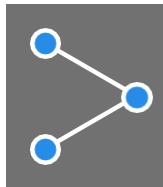
UNIDAD

POLIMORFISMO

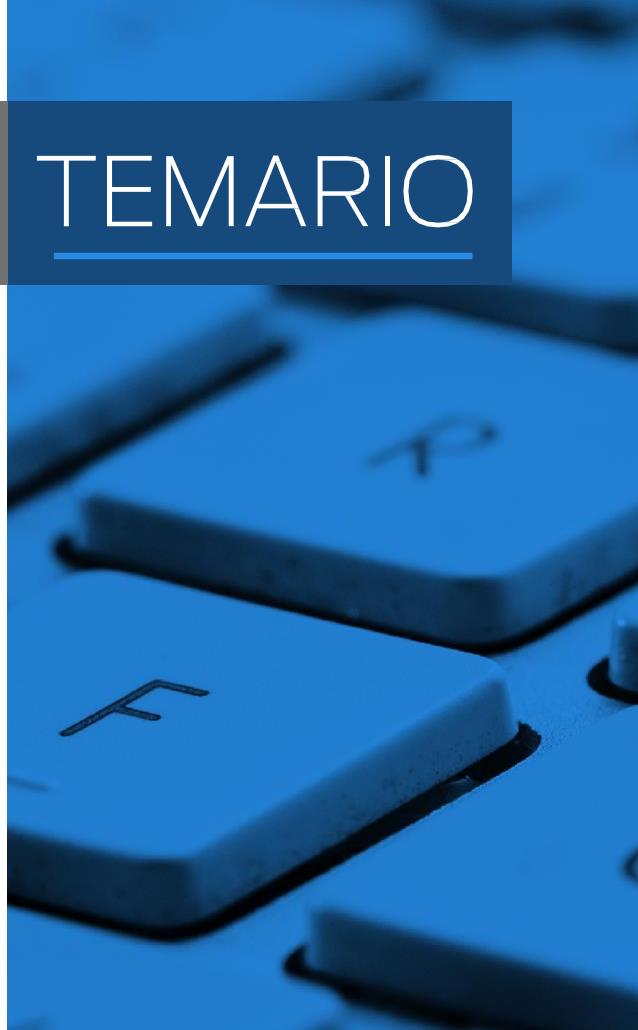


3.1

SOBRECARGA



TEMARIO



3.2

SOBREESCRITURA

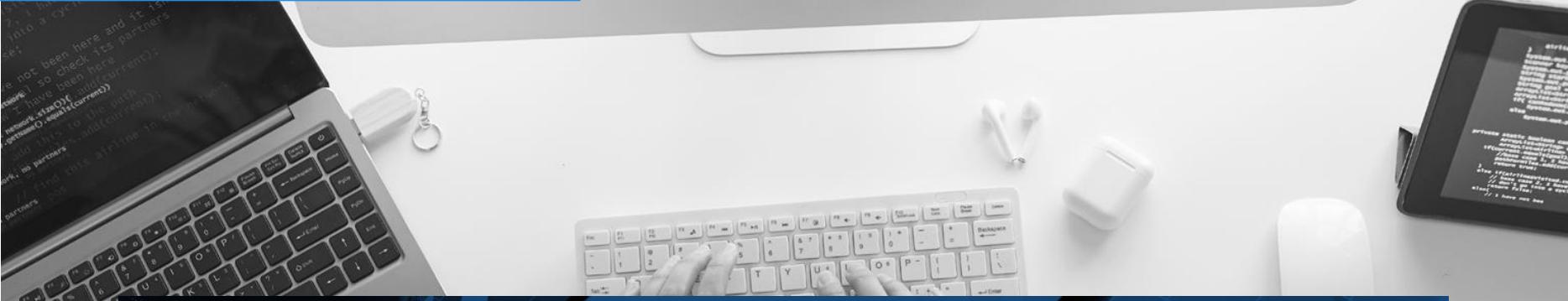




INTRODUCCIÓN

En esta tercera unidad aprenderás más acerca del polimorfismo. Esta es una herramienta esencial, sobre todo cuando se requiere manejar la misma información o datos en distintos objetos. Esto, con el fin de agilizar los procesos.

COMPETENCIAS A DESARROLLAR



El alumno será capaz
de implementar la
sobrecarga de métodos.



El alumno será capaz
de implementar la
sobreescritura de métodos.

VIDEO



Te invitamos a ver el siguiente video:



SOBRECARGA

En la programación orientada a objetos se denomina **polimorfismo** a la capacidad que tienen los métodos para responder, de manera distinta, a partir de su invocación. El polimorfismo se clasifica en dos grandes clases:

- Dinámico
- Estático



SOBRECARGA



- **Polimorfismo dinámico.** También conocido como paramétrico. Es aquel en el que el código no incluye ningún tipo de especificación acerca del tipo de datos sobre el que se trabaja. Así, puede ser utilizado en todo tipo de datos compatible.
- **Polimorfismo estático.** También conocido como *ad hoc*. Es aquel en el que los tipos a los que se aplica el polimorfismo deben ser explicitados y declarados uno por uno antes de poder ser utilizados.

SOBRECARGA

A su vez, el polimorfismo puede dividirse en dos grandes bloques: sobrecarga y sobreescritura.

Por ejemplo, el método que se seleccione puede ser distinto a partir de los valores enviados. A esto se le llama **sobrecarga de métodos**.

Una de las ventajas de la sobrecarga es que permite la solicitud de una operación sin conocer el método que debe ser llamado.



SOBRECARGA



○ **Método sobrecargado.** Son aquellos métodos que pueden ser implementados a partir de diferentes tipos de datos y por ello reaccionan de forma indistinta.

En este sentido, la sobrecarga de métodos no es otra cosa sino la posibilidad de construir varios métodos con el mismo nombre. Estos tienen una relación con la clase a la que pertenece cada uno pero, a pesar de llamarse igual, tienen comportamientos diferentes.

SOBRECARGA

En la sobrecarga de métodos podemos definir una clase con dos métodos con el mismo nombre si los argumentos son distintos.

Son métodos distintos porque, aunque tengan el mismo nombre, tienen distintos argumentos y distinta funcionalidad.

La sobrecarga se utiliza mucho para los constructores.

Este concepto se ejemplifica a continuación:



SOBRECARGA

```
public class Coche {  
    private String color;  
    private int velocidad;  
    private float tamaño;  
  
    public Coche (String color, int velocidad, float tamaño){  
        this.color = color;  
        this.velocidad = velocidad;  
        this.tamaño = tamaño;  
    }  
  
    public void avanzar(){}
    public void avanzar(int metros){}
    public void avanzar(int metros, int velocidad){}
  
  
    public void parar(){}
    public void girarIzquierda(){}
    public void girarDerecha(){}
}
```

SOBRECARGA

Como se puede ver en el ejemplo anterior,
existen tres métodos:

- public void avanzar(){}
- public void avanzar(int metros){}
- public void avanzar(int metros, int velocidad)
 {}

En este caso, los tres se llaman igual; sin embargo, dado que dentro del paréntesis reciben argumentos distintos, cada uno será llamado dependiendo de los argumentos con los cuales sea convocado.



SOBRECARGA

El siguiente ejemplo ilustra cómo se implementan dos métodos sobrecargados:

```
// The area of a rectangle.  
float area(const float& length,  
           const float& width)  
{  
    return length * width;  
}  
  
// The area of a circle.  
float area(const float& radius)  
{  
    constexpr float pi = 3.14159265358979323846F;  
  
    return (pi * radius) * radius;  
}
```

VIDEO



Te invitamos a ver el siguiente video:



SOBREESCRITURA

Un **método virtual** es una función miembro, pública o protegida, de una clase base que puede ser redefinida en cada una de las clases derivadas de esta. A su vez, puede ser accedido mediante un puntero o referencia de la clase base.

El método se declara colocando la palabra clave *virtual* antes de la declaración:

```
virtual<tipo de dato><nombre de la función>(lista de parámetros);
```

SOBREESCRITURA

En programación orientada a objetos, cuando una clase derivada hereda de una clase base un objeto de aquella, este puede ser referido tanto por el tipo de la clase base como por el tipo de la clase derivada.

Cuando un objeto derivado es referido como del tipo de la clase base, el comportamiento de la llamada a la función deseada es ambiguo. Una distinción entre virtual y no virtual sirve para resolver este problema.



SOBREESCRITURA

Una clase que incluya una función virtual es llamada una **clase polimórfica**. Este término también aplica a una clase que hereda una clase base conteniendo una función virtual. A continuación, un ejemplo:

```
#include <iostream>
using namespace std;
class base {
public:
    virtual void quien() {cout << "Base" << endl;} // especificar
    una clase virtual
};
class primera_d : public base {
public:
    // redefinir quien() relativa a primera_d
    void quien() relativa a primera_d
    void quien() {cout <<"Primera derivación" << endl;}
```

SOBREESCRITURA

Continuación:

```
class segunda_d : public base {  
public:  
// redefinir quien relativa a segunda_d  
void quien() {cout <<“Segunda derivación” << endl;}  
};
```

SOBREESCRITURA

Continuación:

```
int main() {  
base obj_base;  
base *p;  
primera_d obj_primera;  
segunda_d obj_segundo;  
p = &obj_base;  
p->quien(); // acceder a quien() en base  
p = &obj_primera;  
p->quien(); // acceder a quien() en primera_d  
P = $obj_segundo;  
p->quien(); // acceder a quien() en segunda_d  
return 0;  
}
```

SOBREESCRITURA



A continuación, examinaremos el anterior programa a detalle, para comprender cómo funciona. En base, la función *quien()* es declarada como virtual. Esto significa que la función puede ser redefinida en una clase derivada. Dentro de ambas, *primera_d* y *segunda_d*, *quien()* es redefinida relativa a cada clase. Dentro de *main()*, 4 variables son declaradas: *obj_base* (objeto de tipo base), *p* (puntero a objetos tipo base), *obj_primera* y *obj_segunda*, que son objetos de 2 clases derivadas. A continuación, *p* es asignada con la dirección de *obj_base*; y la función de *quien()* es llamada. Dado que *quien()* es declarada como virtual, C++ determina en tiempo de ejecución cuál versión será referenciada por el tipo del objeto apuntado por *p*.

SOBREESCRITURA



En este caso, *p* apunta a un objeto del tipo base, así que es la versión de *quien()*, declarada en *base*, la que es ejecutada. A continuación, se le asigna a *p* la dirección de *obj_primer*. Recuerda que un puntero de la clase base puede referirse a un objeto de cualquier clase derivada. Ahora, cuando *quien()* es llamada, C++ nuevamente comprueba para ver qué tipo de objeto es apuntado por *p* y, basado en su tipo, determina cuál versión de *quien()* llamar. Como *p* apunta a un objeto del tipo *obj_primer*, esa versión de *quien()* es usada.

De ese mismo modo, cuando *p* es asignado con la dirección de *obj_segunda*, la versión de *quien()* declarada en *segunda_d* es ejecutada.

SOBREESCRITURA

Un **método virtual puro** es una función virtual que necesita ser implementada por una clase derivada que no sea abstracta. Las clases que contienen métodos virtuales puros son denominadas abstractas. Estas no pueden ser instanciadas directamente.

Además, una subclase de una clase abstracta solo puede ser instanciada directamente si todos los métodos virtuales puros han sido implementados por esa clase o una clase padre.



SOBREESCRITURA

En C++, los métodos virtuales puros son declarados utilizando una sintaxis especial (**=0**), como se exemplifica a continuación:

```
class B {  
    virtual void una_función_virtual_pura() = 0;  
};
```

SOBREESCRITURA

La declaración de un método virtual puro ofrece solo la cabecera del método. Normalmente no se ofrece una implementación de la función virtual pura en una clase abstracta, pero es posible. Toda clase hija no abstracta continúa necesitando redefinir el método, pero la implementación ofrecida por la clase abstracta puede ser llamada de esta forma:

```
void Abstracta::virtual_pura() {  
    // haz algo  
}  
class Hija : Abstracta {  
    virtual void virtual_pura(); // ya no es abstracta; puede  
    Instanciada  
};
```



SOBREESCRITURA

```
void Hija::virtual_pura() {  
Abstract::virtual_pura(); //se ejecuta la implementación  
de la clase abstracta  
}
```

El compilador sabe a qué implementación del método llamar en tiempo de ejecución creando una tabla de punteros a todos los métodos virtuales de una clase, llamada *vtable* o tabla virtual.

Otra alternativa es agregar métodos virtuales (a la clase base) sin ningún tipo de implementación. Estos métodos se declaran asignándoles el valor cero. Por ejemplo:

```
class Animal { public:virtual void Habla() = 0; };
```

SOBREESCRITURA

```
lass Animal {  
public:  
virtual void Habla() {  
cout << "?" << endl;  
} };  
class Perro : public Animal{  
public:  
void Habla() {  
cout << "Guau" << endl;  
}  
};  
class Gato:public Animal {  
public:  
void Habla () {  
cout << "Miau" << endl; }  
};
```

```
int main() {  
Animal *a1 = new Perro;  
Animal *a2 = new Gato;  
a1->Habla();  
a2->Habla();  
delete a1;  
delete a2;  
return 0;  
}
```

VIDEO



Te invitamos a ver el siguiente video:



ACTIVIDAD 2

Te invitamos a realizar la siguiente actividad:

Presiona el botón para descargar la actividad:



Presiona el botón para entregar la actividad:



El polimorfismo es una característica que no puede ser vista de forma aislada. Por lo tanto, funciona únicamente de forma conjunta, como una parte de una gran cuadro de relaciones entre clases

Para utilizar el polimorfismo y, por lo tanto, las técnicas orientadas a objetos en los programas, hay que ampliar la visión de la programación para incluir no solo miembros y mensajes entre clases individuales, sino también sus puntos en común, así como sus relaciones.

Aunque requiere un esfuerzo significativo, este es recompensado gracias a que se consigue mayor velocidad en el desarrollo, mejor organización de código, programas extensibles y mayor mantenibilidad.

CONCLUSIÓN



```
function (ko, datacontext) {}  
<div style="background-image:url('1px')>  
background .text - todoitem;  
height .text - :200px;">  
<p>The image can be tiled across the background.  
</div>  
  
|| persisted properties  
  
<html> <p style="font-weight:bold;">HTML font codes are:  
<html> <body style="background-color:yellow">  
<html> <.todolistid = data.todoId;  
  
|| Non - persisted properties  
<html> <errorMessage = ko.observable();
```

¡FELICIDADES!

Acabas de concluir la [tercera unidad](#) de tu curso *Lenguajes de Programación II*. Te invitamos a finalizar este esfuerzo realizando el examen parcial correspondiente. Para ello, debes regresar a la pantalla principal y dar clic en *Presentar examen*.

4

UNIDAD

ENCAPSULAMIENTO



4.1

MODIFICADORES DE ACCESO

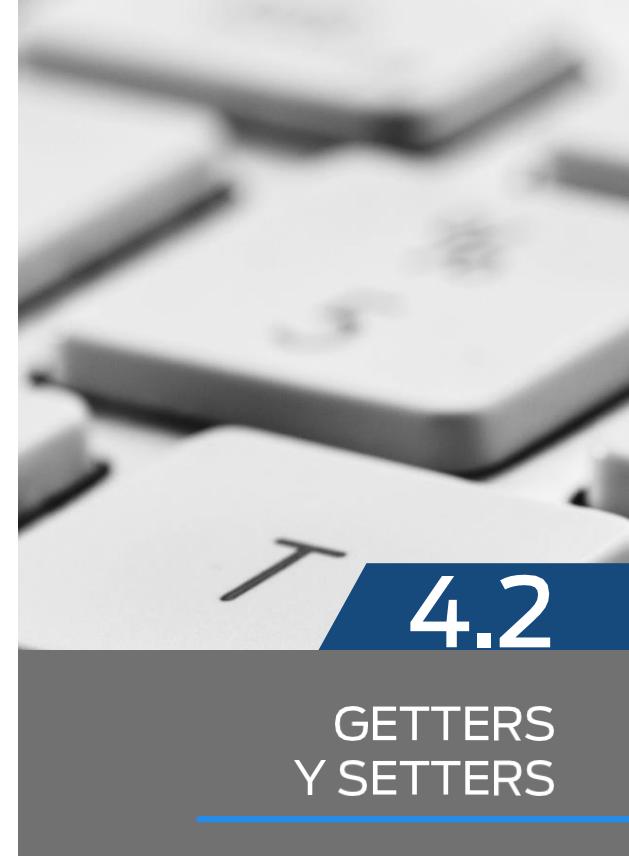


TEMARIO



4.2

GETTERS Y SETTERS





INTRODUCCIÓN

En esta cuarta unidad aprenderás sobre cómo gestionar el acceso a los miembros de una clase. Esto te permitirá ofrecer un mayor control, seguridad y privacidad en tus sistemas de información. Para ello, veremos los modificadores de acceso *public*, *private* y *protected*. A su vez, se implementarán tanto *getters* como *setters*.

COMPETENCIAS A DESARROLLAR





El alumno será capaz de comprender las diferencias entre los modificadores de acceso.



El alumno será capaz de implementar el acceso a miembros de clase mediante *getters* y *setters*.

VIDEO



Te invitamos a ver el siguiente video:





MODIFICADORES DE ACCESO

En C++, los miembros de clase pueden llegar a tener uno de los siguientes tres modificadores de acceso: *public*, *private*, o *protected*.

Las mejores prácticas sugieren que se ordenen los componentes de la clase de acuerdo a sus modificadores de acceso. Es decir, primero los *public*, después los *protected*, y al final los *private*.

MODIFICADORES DE ACCESO

Cuando se define el especificador de acceso se debe tener en cuenta la siguiente correspondencia:

Modificador	Corresponde a:
Public	Todos los miembros públicos de la clase base se tratarán como miembros públicos de sus clases derivadas. De igual manera, pueden ser accedidas desde cualquier parte del programa

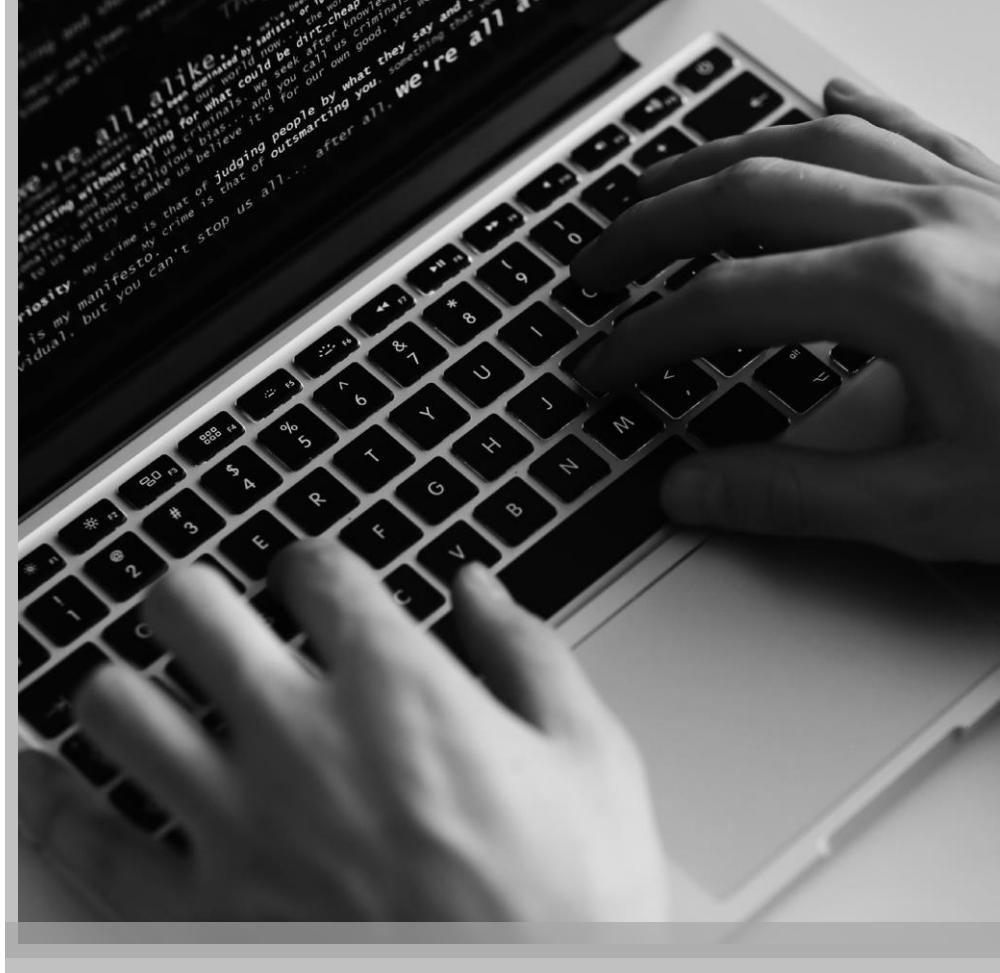
MODIFICADORES DE ACCESO

Modificador	Corresponde a:
Protected	Los miembros protegidos de la clase base se tratan como protegidos de sus clases derivadas. En particular, permiten esconder datos seleccionados y la implementación de sus detalles.
Private	Los miembros privados de la clase base no son accesibles desde las clases derivadas.

MODIFICADORES DE ACCESO

Un caso particular es el *protected*. Este es el punto intermedio entre el *public* y el *private*. Los miembros clasificados como *protected* se comportan como *public* para los métodos de la clase derivada y amigas, y como *private* para el resto de las funciones.

En el caso de que se deje sin especificar el modificador de acceso, el *default* se vuelve *private* para las clases y *public* para las estructuras.



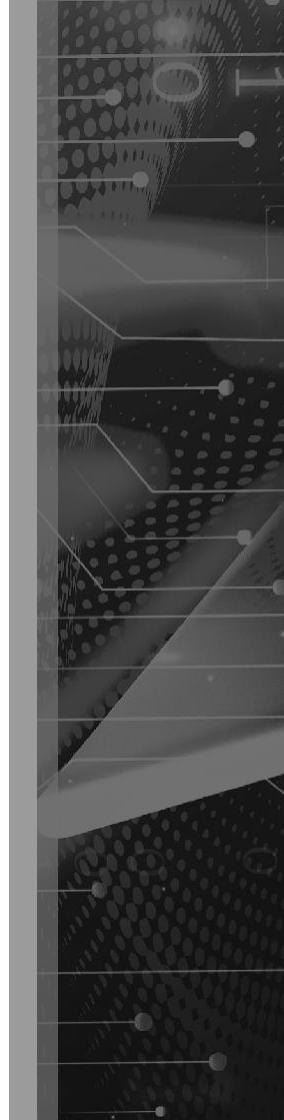
MODIFICADORES DE ACCESO

```
class Vehiculo {  
    .....  
    protected :  
        int peso;  
};  
class V_Tierra : public Vehiculo  
{  
    .....  
    float MomentoInercia () const {  
        return velocidad * peso;  
    }  
    .....  
    private:  
        float velocidad;  
};
```

MODIFICADORES DE ACCESO

Este es un ejemplo de la **implementación de los tres modificadores de acceso:**

```
class led_base {  
public:  
    virtual void toggle() = 0; // Pure abstract.  
    virtual ~led_base() { } // Virtual destructor.  
    // Interface for querying the LED state.  
    bool state_is_on() const { return is_on; }  
protected:  
    bool is_on;  
    // A protected default constructor.  
    led_base() : is_on(false){}  
private:  
    // Private non-implemented copy constructor.  
    led_base(const led_base&) = delete;  
    // Private non-implemented copy assignment operator.  
    const led_base& operator=(const led_base&) = delete;  
}
```



MODIFICADORES DE ACCESO



En el código anterior, la interfaz pública cuenta con dos métodos virtuales. Por tanto, clases derivadas deberán sobreescribirlos y podrán accederse desde todo el programa.

Por su parte, el constructor es protegido. Por tanto, solamente clases derivadas podrán acceder a él. Lo cual tiene sentido porque ninguna clase que no sea *led* debería poder crear instancias de esta clase.

GETTERS Y SETTERS



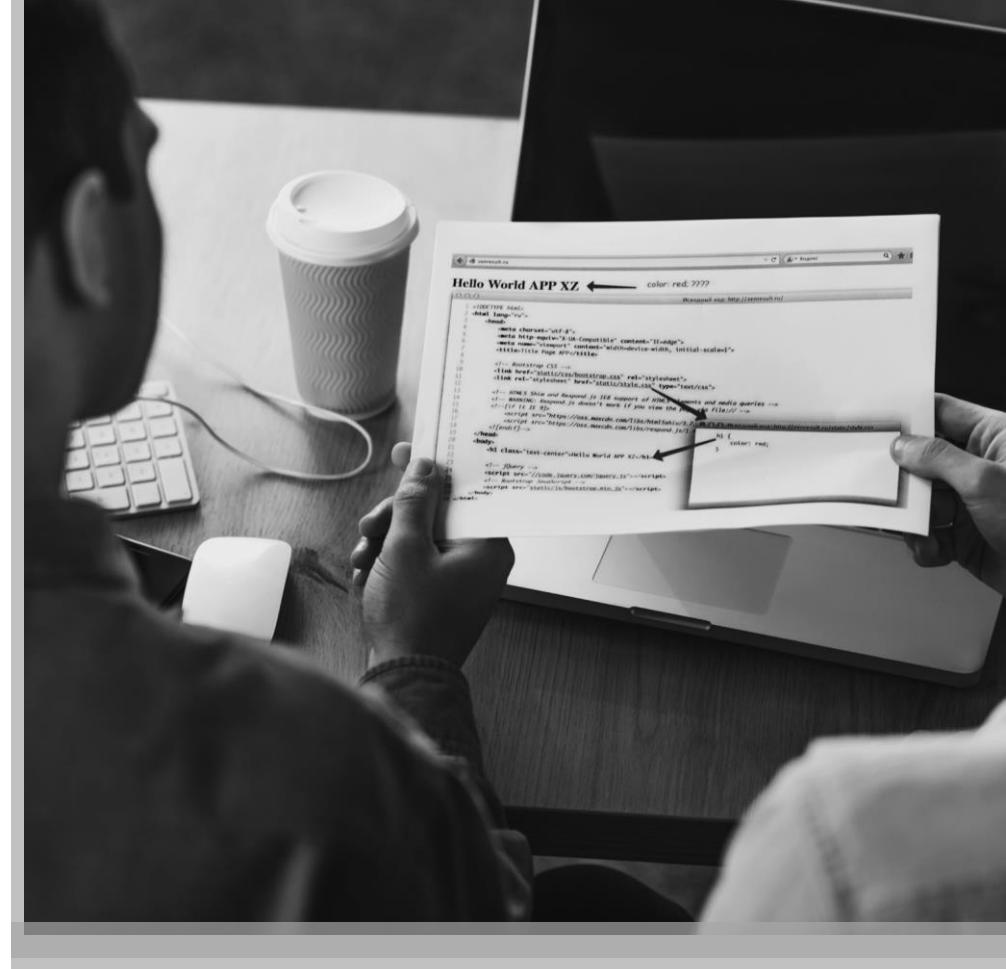
El **encapsulamiento** es una técnica que permite localizar y ocultar los detalles de un objeto. La encapsulación previene que un objeto sea manipulado por operaciones distintas de las definidas.

La encapsulación es como una caja negra que esconde los datos. Solamente permite acceder a ellos de forma controlada.

GETTERS Y SETTERS

Las principales razones técnicas para la utilización de la encapsulación son:

- Mantener a salvo los detalles de representación, si solo nos interesa el comportamiento del objeto.
- Modificar y ajustar la representación a mejores soluciones algorítmicas o a nuevas tecnologías de software.



GETTERS Y SETTERS



La razón de ser del encapsulamiento es, principalmente, limitar o denegar el acceso directo a la variable u objeto. Si se utiliza un método para acceder a ella podrás, dentro de este, realizar ciertas validaciones o establecer un comportamiento en particular cuando se modifica o se intenta modificar su valor.

La encapsulación define los niveles de acceso para elementos de determinada clase. Estos niveles definen los derechos de acceso para los datos.

GETTERS Y SETTERS

Los métodos: *get* y *set* usualmente se definen para el acceso a una variable privada de la clase, para obtener su estado actual (valor) y para modificarlo respectivamente:

- Un método **get**, por lo general, de una sola línea de código, la cual retorna el valor actual de la variable privada.
- Un método **set**, por su parte, consta de por lo menos una línea de código que modifica directamente el valor de la variable privada.



GETTERS Y SETTERS

Un ejemplo de ellos se presenta en el siguiente código:

```
#include <iostream>
using namespace std;

Class Mascota {
    private:
        int edad;

    public:
        void setEdad(int e)
    { edad = e;}
        int getEdad()
    {return edad;}
};
```

```
int main()
{
    Mascota obj;
    obj.setEdad(3);
    cout << obj.getEdad();
    return 9;
}
```

GETTERS Y SETTERS

En el ejemplo anterior podemos identificar lo siguiente:

- Para que la edad de la mascota esté protegida, esta se declara con el modificador de acceso *private*. De esta manera, solamente podrá ser vista por métodos de la clase.
- Esto crea el problema de que no podrá ser accedido desde fuera de la clase, por ejemplo, el método *main()*.
- Por tanto, se crea un método para enviarle valores y otro para obtener el valor actual.

GETTERS Y SETTERS

- Así, ambos métodos se declaran públicos en la clase *Mascota*.
- De tal manera, el método *main()* puede utilizar uno de esos métodos para enviar un valor a la edad de la mascota (*i.e.*, *setEdad()*).
- Posteriormente, se obtiene el valor de la edad de la mascota a través de otro método (*i.e.*, *getEdad()*)..
- Finalmente, se imprime la edad y es posible ver que es la que se le envió.



VIDEO



Te invitamos a ver el siguiente video:



El encapsulamiento permite tener control sobre los datos que manejamos en nuestros programas. Para incrementar la seguridad de los datos se utilizan modificadores de acceso. Sin embargo, al quedar protegidos por el modificador *private*, algunos datos quedan aislados de procesos que pudieran llegar a ocuparlos.

Por ello, existen los métodos conocidos como *getters* y *setters*. Este tipo de métodos permiten gestionar el acceso a los datos de una manera controlada y a partir de puntos específicos del programa. Además, como son controles planificados, se puede evaluar el procedimiento de acceso.

CONCLUSIÓN



```
function (ko, datacontext) {}  
<div style="background-image:url('1px')>  
    background .text - todoitem;  
    height .text - :200px;">  
    <p>The image can be tiled across the background.  
</div>  
  
|| persisted properties  
  
<html> <p style="font-weight:bold;">HTML font code is displayed  
<html> <body style="background-color:yellow">  
<html> <.todolistid = data.todoId;  
  
|| Non - persisted properties  
<html> <errorMessage = ko.observable();
```

¡FELICIDADES!

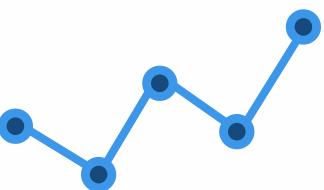
Acabas de concluir la cuarta unidad de tu curso *Lenguajes de Programación II*. Te invitamos a finalizar este esfuerzo realizando el examen parcial correspondiente. Para ello, debes regresar a la pantalla principal y dar clic en *Presentar examen*.

BIBLIOGRAFÍA

- Deitel, H.M. & Deitel P.J. (2014). *Cómo programar en C++*. Prentice Hall.
- Eckel, B. (2012). Pensar en C++. Volumen 1. Kormanyos, C. (2015) *Real-Time C++: Efficient Object-Oriented and Template Microcontroller Programming*. Springer.



PROYECTO FINAL



PROYECTO FINAL

Te invitamos a realizar el siguiente proyecto final:

Presiona el botón para descargar el proyecto final:



Presiona el botón para entregar el proyecto final:

