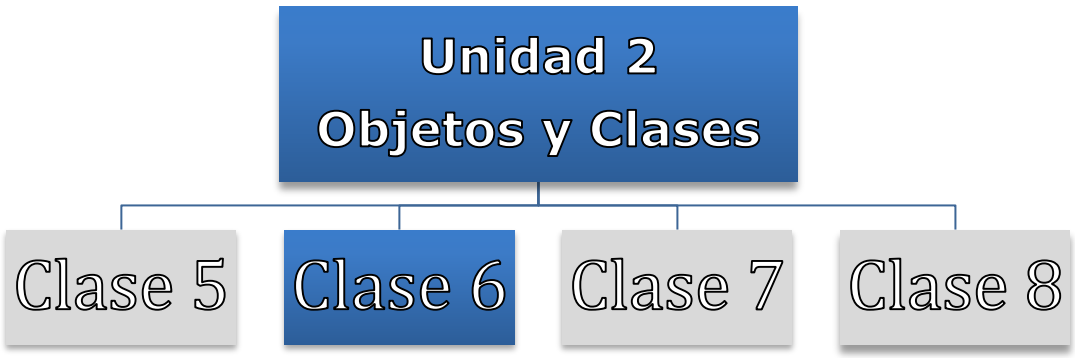
PROGRAMACIÓN ORIENTADA A OBJETOS



Docente titular y autor de contenidos: Prof. Ing. Darío Cardacci

Presentación

En la clase seis haremos énfasis en los distintos tipos de clase y para qué se usan y de esta manera poder potenciar nuestros desarrollos.

También analizaremos la relación de herencia que se desprende la jerarquía "es-un", la sobrescritura de métodos y el polimorfismo.

Esperamos que luego de analizar estos temas Ud. pueda percibir el aporte que los mismos les otorgan a los desarrollos orientados a objetos.

Por todo lo expresado hasta aquí es que esperamos que usted, a través del estudio de esta unidad logre:

- Comprender la noción de herencia.
- Aplicar la sobrescritura de métodos.
- Analizar y reconocer las relaciones los escenarios que pueden aplicar polimorfismo.
- Desarrollar métodos polimórficos.

Los siguientes **contenidos** conforman el marco teórico y práctico de esta unidad. A partir de ellos lograremos alcanzar el resultado de aprendizaje propuesto. En negrita encontrará lo que trabajaremos en la clase 6.

- Eventos. Suscripción a eventos. Suscripción a eventos utilizando el IDE. Suscripción a eventos mediante programación. Suscripción a eventos mediante métodos anónimos. Publicación de eventos. Desencadenar eventos.
- Modificadores de acceso.

- **Tipos de clases. Clases abstractas, selladas y estáticas. Miembros estáticos en clases estáticas.**
- **Relaciones básicas entre clases. "Generalización-Especialización", "Parte de".**
- **Relaciones derivadas entre clases. Herencia. Herencia simple. Herencia múltiple. Teoría de Tipos. Tipos anónimos.**
- **Sobrescritura de métodos. Métodos virtuales. Polimorfismo.**
- Agregación. Simple y con contención física.
- Asociación y relación de Uso.
- Elementos que determinan la calidad de una clase: acoplamiento, cohesión, suficiencia, compleción y primitivas.
- Relaciones entre objetos: enlace y agregación.
- Acceso a la clase base desde la clase derivada.
- Acceso a la instancia actual de la clase.

A continuación, le presentamos un detalle de los contenidos y actividades que integran esta clase. Usted deberá ir avanzando en el estudio y profundización de los diferentes temas, realizando las lecturas requeridas y elaborando las actividades propuestas, algunas de desarrollo individual y otras para resolver en colaboración con otros estudiantes y con su profesor tutor.

Índice de contenidos y Actividades

1. Herencia y Teoría de Tipos.

Lectura requerida

- Deitel Harvey M. y Deitel Paul J. Cómo programar C#. Segunda Edición Pearson Educación. 2007. Capítulo X.
- Deitel Harvey M. y Deitel Paul J. Cómo programar C#. Segunda Edición Pearson Educación. 2007. Capítulo XI.
- https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/inheritance
- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/

2. Sobrescritura y Polimorfismo

Lectura requerida

- Deitel Harvey M. y Deitel Paul J. Cómo programar C#. Segunda Edición Pearson Educación. 2007. Capítulo XI.
- https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/override

Para el estudio de estos contenidos usted deberá consultar la bibliografía que aquí se menciona:

BIBLIOGRAFÍA RECOMENDADA

- Deitel Harvey M. y Deitel Paul J. Cómo programar C#. Segunda Edición Pearson Educación. 2007.

Links a temas de interés

Desarrollo .NET

https://docs.microsoft.com/en-us/dotnet/csharp/index

https://docs.microsoft.com/en-us/dotnet/standard/get-started

Lo/a invitamos ahora a comenzar con el estudio de los contenidos que conforman esta unidad.

1. Herencia y Teoría de Tipos.

La **herencia** es una característica de los lenguajes de programación orientados a objetos. Esta característica permite definir una clase base, que proporciona una estructura y una funcionalidad específica (datos y comportamiento). Lo que posee la clase base y es heredado a las clases derivadas pasan a formar parte de la estructura y las funcionalidades de estas.

Las clases derivadas pueden amplíar o modificar el comportamiento de su clase base. C# y .Net solo admiten la herencia simple. Esto significa que una clase derivada solo puede heredar de una única clase base. Sin embargo, la herencia es transitiva, lo que le permite definir una jerarquía de herencia para un conjunto de tipos. En otras palabras, el tipo Z puede heredar del tipo Y, que hereda del tipo X. Dado que la herencia es transitiva, los miembros de tipo X están disponibles para el tipo Z.

No todos los miembros de una clase base son heredados por sus clases derivadas. Los siguientes miembros no se heredan:

- Constructores estáticos, que inicializan los datos estáticos de una clase.
- Constructores de instancias, a los que se llama para crear una nueva instancia de la clase. Cada clase debe definir sus propios constructores.
- Finalizadores, llamados por el recolector de elementos no utilizados en tiempo de ejecución para destruir instancias de una clase.

Si bien las clases derivadas heredan todos los demás miembros de su clase base, que dichos miembros estén o no visibles depende de su accesibilidad. La accesibilidad que el miembro posea en la clase base, afecta su visibilidad en las clases derivadas. Por ejemplo:

- Los miembros privados solo son visible en las clases derivadas que están anidadas en su clase base. De lo contrario, no son visibles en las clases derivadas.
- Los miembros protegidos solo son visibles en las clases derivadas.

- Los miembros internos solo son visibles en las clases derivadas que se encuentran en el mismo ensamblado que la clase base. No son visibles en las clases derivadas ubicadas en un ensamblado diferente al de la clase base.
- Los miembros públicos son visibles en las clases derivadas y forman parte de la interfaz pública de dichas clases. Los miembros públicos heredados se pueden llamar como si se hubieran definido en la clase derivada.

En el Ejemplo **Ej0016** se puede observar como se programa una herencia. En este caso la clase derivada, llamada **ClaseDerivada,** hereda de la clase base denominada **ClaseBase** de la siguiente forma: public class ClaseDerivada : ClaseBase.

```
public class ClaseBase
{
    1 referencia
    public ClaseBase() { MessageBox.Show("Constructor de ClaseBase"); }
    private string A = "Miembro private A";
    protected string B = "Miembro Protected B";
    internal string C = "Miembro  internal C";
    public string D = "Miembro public D";

}
3 referencias
public class ClaseDerivada : ClaseBase
{
    1 referencia
    public ClaseDerivada() : base()
    { MessageBox.Show("Constructor de ClaseDerivada"); }
    1 referencia
    public string Bderivada() {return this.B; }
    1 referencia
    public string Bderivada2() { return base.B; }
    1 referencia
    public string Cderivada() { return this.C; }
    1 referencia
    public string Cderivada2() { return base.C; }
    1 referencia
    public string Dderivada() { return this.D; }
    1 referencia
    public string Dderivada2() { return base.D; }
}
```

Ej0016

Más abajo se observa la implementación y uso de la clase derivada.

```
private void button1_Click(object sender, EventArgs e)
{
    ClaseDerivada CD = new ClaseDerivada();
    // Consume el campo B heredado a la clase derivada a través del método Bderivada.
    MessageBox.Show(CD.Bderivada());
    // Consume el campo B desde la clase derivada a través del método Bderivada2 accediendo
    // a la clase base.
    MessageBox.Show(CD.Bderivada2());
    // Consume el campo C heredado a la clase derivada a través del método Cderivada.
    MessageBox.Show(CD.Cderivada());
    // Consume el campo C desde la clase derivada a través del método Cderivada2 accediendo
    // a la clase base.
    MessageBox.Show(CD.Cderivada2());
    // Consume el campo D heredado a la clase derivada a través del método Dderivada.
    MessageBox.Show(CD.Dderivada());
    // Consume el campo D desde la clase derivada a través del método Dderivada2 accediendo
    // a la clase base.
    MessageBox.Show(CD.Dderivada2());
    // Consume el campo C heredado de la clase base.
    MessageBox.Show(CD.C);
    // Consume el campo D heredado de la clase base.
    MessageBox.Show(CD.D);
}
```

Ej0016

En .NET existe una **herencia implícita**. Además de la herencia simple que cualquier clase derivada puede tener, todos los tipos de .NET heredan implicitamente de un tipo denominado **Object**. Esto garantiza que la funcionalidad común que implementa **Object** está disponible para cualquier tipo. En el ejemplo **Ej0017** se puede observar la clase **Acceso** que no define ningún miembro.

```
public class Acceso { }
```

Ej0017

Sin embargo al consultarla mediante **reflection** se puede observar que expone los siguiente miembros:

Ej0017

Estos miembros son los que se heredaron implísitamente de **Object.**

Normalmente, la herencia se usa para expresar una relación "es - un" entre una clase base y una o varias clases derivadas, donde las clases derivadas son versiones especializadas de la clase base.

Este tipo de relacion permite que la clase derivada defina su propio tipo pero también implementa el tipo de su clase base. Por ejemplo, si tenemos una clase base denominada **Vehiculo** y una clase derivada denominada **Auto,** se puede expresar que **Auto "es - un" Vehiculo.** Esto tiene multiples consecuencias, pero una muy significativa es que las instancias de **Vehiculo** son de tipo Vehiculo y pueden ser apuntadas por variables de tipo vehículo. Pero las instancias de **Auto** son de tipo Auto y podrán ser apuntadas por variables de tipo Auto o por variables de tipo Vehiculo, debido a que como ya dijimos un **Auto "es - un" Vehiculo**.

Cuando la instancia de **Auto** sea apuntada por una variable de tipo Auto, si se accede a su interfaz se observará la que define **Auto.** Pero si la misma instancia de **Auto** es apuntada por una variable de tipo Vehiculo, se observará la interfaz que define el tipo **Vehiculo,** a pesar que el objeto es el mismo.

Esto puede observarse en el ejemplo **Ej0018.** La clase **Vehiculo** implementa una propiedad denominada **Marca.** La clase **Auto** hereda de **Vehiculo** y además implementa las propiedades **Modelo** y **Precio.**

```
public class Vehiculo
{
    4 referencias
    public string Marca { get; set; }
}

2 referencias
public class Auto : Vehiculo
{
    3 referencias
    public string Modelo { get; set; }
    3 referencias
    public decimal Precio { get; set; }
}
```

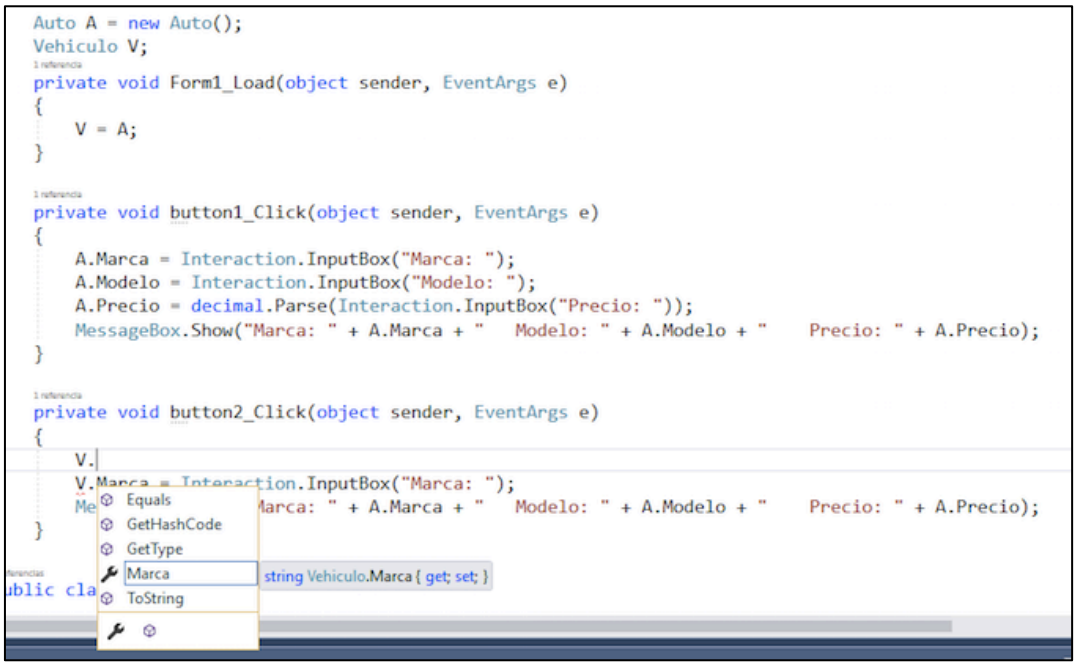
Ej0018

```
Auto A = new Auto();
Vehiculo V;
1 referencia
private void Form1_Load(object sender, EventArgs e)
{
    V = A;
}

1 referencia
private void button1_Click(object sender, EventArgs e)
{
    A.
    A.Marca = Interaction.InputBox("Marca: ");
    A.                    tion.InputBox("Modelo: ");
    A.                    .Parse(Interaction.InputBox("Precio: "));
    Me                    Marca: " + A.Marca + "   Modelo: " + A.Modelo + "    Precio: " + A.Precio);
}
    Equals
    GetHashCode
    GetType
    Marca
    Modelo
    Precio
    ToString
    string Vehiculo.Marca { get; set; }

1 referencia
privat                    lick(object sender, EventArgs e)
{
    V.                    tion.InputBox("Marca: ");
    MessageBox.Show("Marca: " + A.Marca + "   Modelo: " + A.Modelo + "    Precio: " + A.Precio);
}
```

Ej0018

También se puede observar que se instancia un **Auto** que es apuntado por la variabe **A**, que es de tipo **Auto.** Se declara la variables **V** del tipo **Vehiculo.**

Luego se establece que **V=A,** osea que **V** apunta al mismo objeto que apunta **A.** Si colocamos **A** más punto (**A.**) veremos la interfaz que define **Auto.** En ella se pueden observar las propiedades **Marca, Modelo y Precio.** Sin embargo, si colocamos **V** más punto (**V.**) veremos la interfaz de **Vehiculo** que solo da visibilidad a la propiedad **Marca.**



Ej0018

Lo observado se debe a que **Auto** hereda de **Vehiculo**. Mientras que **Vehiculo** solo define la propiedad **Marca, Auto** la hereda y además de especializa incorporando **Modelo** y **Precio** como propiedades propias.

Si se ejecuta el código del ejemplo se podrá observar que desde la variable **A** que apunta a la instancia de **Auto** se puede modificar y/o consultar a las tres propiedades. Si utilizamos la variable **V** de tipo **Vehiculo,** que apunta al mismo objeto, solo se puede modificar y/o consultar la propiedad **Marca**. A pesar de ello la marca cargada desde la variable **V** afecacta el estado del único objeto que tenemos y es debido a ello que cuando consultamos la propiedad **Marca** desde la variable **A** el valor se ve alterado.

Se debe tener en cuenta que la relación **"es - un"** también expresa la relación entre un tipo y una instancia específica de ese tipo. Esto lo podemos traducir como que todo objeto posee un tipo.

2. Sobrescritura y Polimorfismo

La **sobrescritura** se utiliza para modificar la implementación abstracta o virtual de un método, propiedad, indizador o evento heredado. El modificador que se utiliza para lograrlo es **override.**

Un método afectado con el modificador **override** proporciona una nueva implementación de un miembro que se hereda de una clase base. El método que se anula mediante la declaración **override** se conoce como el método base anulado. El método base anulado debe tener la misma firma que el método que posee el modificador **override.**

Los metodos **estáticos** y **no virtuales** no se pueden anular utilizando el modificador **override.** El método base debe ser **virtual, abstract** u **override** para ser anulado por **override.**

Una declaración con el modificador **override** no puede cambiar la accesibilidad del método base **virtual**. Tanto el metodo afectado con **override** como el método **virtual** deben tener el mismo modificador de nivel de acceso.

El ejemplo **Ej0019** muestra como realizar una sobrescritura.

```
        private void button1_Click(object sender, EventArgs e)
        {
            Cuadrado C = new Cuadrado(int.Parse(Interaction.InputBox("Lado: ")));
            MessageBox.Show("Area del cuadrado: " + C.Area().ToString());
        }
    }
    1 referencia
    abstract class Forma
    {
        2 referencias
        abstract public int Area();
    }
    3 referencias
    class Cuadrado : Forma
    {
        int lado = 0;
        1 referencia
        public Cuadrado(int n) { lado = n; }
        // Como Cuadrado hereda de Forma y en Forma está el método abstracto Area
        // Aquí se debe implementar obligatoriamente con override para no generar
        // un error en tiempo de compilación.
        2 referencias
        public override int Area() { return lado * lado; }
    }
```

Ej0019

Se puede observar la clase abstracta **Forma,** la cual posee un método también abstracto denominado **Area.** La clase **Cuadrado** hereda de **Forma** y tiene la obligación de implementar el método abstracto antes mencionado. Lo implementa y sobrescribe, dándole una lógica que permite calcular el área de un cuadrado.

Observemos el mismo ejemplo pero utilizando un método virtual en lugar de un método abstracto.

```
abstract class Impuesto
{
    7 referencias
    public decimal Importe { get; set; }
    3 referencias
    virtual public decimal ImpuestoEnPesos() { return this.Importe * 0.10m; }
    4 referencias
    abstract public decimal TotalAPagar();
}
2 referencias
class ImpuestoComun : Impuesto
{
    4 referencias
    override public decimal TotalAPagar() { return Importe + ImpuestoEnPesos(); }
}
2 referencias
class ImpuestoEspecial : Impuesto
{
    3 referencias
    override public decimal ImpuestoEnPesos() { return this.Importe * 0.20m; }
    1 referencia
    private decimal TasaAdicional() { return this.Importe * 0.01m; }
    4 referencias
    override public decimal TotalAPagar() { return Importe + ImpuestoEnPesos() + TasaAdicional(); }
}
```

Ej0020

En el ejemplo **Ej0020** se puede observar una **clase abstracta** denominada **Impuesto** que oficia de clase base de las clases derivadas **ImpuestoComun** e **ImpuestoEspecial.** La clase **Impuesto** define una propiedad pública llamada **Importe**, un **método virtual** denominado **ImpuestoEnPesos** y finalmente un **método abstracto** llamado **TotalAPagar.**

El **metodo virtual ImpuestoEnPesos** implementa código, en este caso retorna el 10% del Importe. Si observamos las clases derivadas, se puede ver que en el caso de **ImpuestoComun** no se anula la implementación heredada de **ImpuestoEnPesos**. Esto se debe a que como el método está marcado con **virtual**, este modificador da la posibilidad de anular la implementación o utilizarla como se ha heredado. Claramente la clase derivada **ImpuestoComun** ha hecho esto último. En el caso de la clase derivada **ImpuestoEspecial,** se ha optado por anular la implementación heredada. Para ello se utilizó el modificador **override.** La nueva implementación realiza un calculo equivalente al 20% del importe.

Un aspecto muy sgnificativo en orientación a objetos es el **polimorfismo**. El **polimosfismo** se da cuando un método heredado por dos o más sub clases, poseen implementaciones diferentes. En el ejemplo **Ej0021/Ej0022** se puede observar la clase **Producto** que posee una propiedad **Precio** y un método **PrecioConDescuento.** El método es abstracto y se implementa de distinta manera en las subclases **ProductoNacional** y **ProductoImportado.**

```
abstract public class Producto
    4 referencias
{   public decimal Precio { get; set; }
    3 referencias
    abstract public decimal PrecioConDescuento();
}
1 referencia
public class ProductoNacional : Producto
{
    3 referencias
    public override decimal PrecioConDescuento()
    {
        return this.Precio * 0.90m;
    }
}
1 referencia
public class ProductoImportado : Producto
{
    3 referencias
    public override decimal PrecioConDescuento()
    {
        return this.Precio * 0.80m;
    }
}
```

Ej0021/Ej0022

Las clases **ProductoNacional** y **ProductoImportado** poseen en comun el tipo **Producto** ya que ambas heredan de él. Esto hace posible que un objeto del tipo **ProductoNacional** como un objeto del tipo **ProductoImportado** puedan ser apuntados por una variable de tipo **Producto**, pues de hecho al heredar de la clase **Producto,** ambas instancias gozan de esa posibilidad pues por herencia son productos. También podemos extender la idea a que un parámetro de tipo **Producto** pueda recibir objetos que sean instancia de **Producto** (siempre que pruducto no sea una clase abstracta) u objetos que sean instancias de subclases que hayan heredado de **Producto.**

```
private void CalculaDescuento(Producto P)
{
    P.Precio=decimal.Parse(Interaction.InputBox("Ingrese el precio del producto nacional: "));
    MessageBox.Show("El precio sin descuento es: " + P.Precio + "\n" +
                    "El precio con descuento es: " + P.PrecioConDescuento().ToString());
}
```

Ej0021/Ej0022

La función **CalculaDescuento** posee el parámetro **P**, y allí se puede enviar cualquier objeto que sea una instancia de alguna subclase que hereda de **Producto.** También podría recibir una instancia de **Producto,** si esta clase no fuera abstracta. En el ejemplo **Ej0021/Ej0022,** el código de las funciones **button1_Click** y **button2_Click** le envían al parámetro **P** de la función **CalculaDescuento,** un objeto **ProductoNacional** y un **ProductoInternacional** respectivamente.

```
private void button1_Click(object sender, EventArgs e)
{
    this.CalculaDescuento( new ProductoNacional());
}

1 referencia
private void button2_Click(object sender, EventArgs e)
{
    this.CalculaDescuento(new ProductoImportado());
}
```

Ej0021/Ej0022

La función simplemente toma el producto a través del parámetro **P** y ejecuta el método **PrecioConDescuento.** El código que se ejecuta dependerá exclusivamente del objeto enviado. En términos prácticos cuantas más clase tengamos que hereden de **Producto** e implementen de distinta manera el método **PrecioConDescuento**, tendremos más formas distintas para el funcionamiento de la función **CalculaDescuento.**

Es interesante observar que se está logrando que la función actúe de distintas formas, sin necesidad de evaluar con una instrucción

condicional del tipo if, if…else o alguna derivada de ella. Esto acarreaa múltiples beneficios. Quizá el más importante es que podemos hacer que una función programada pueda ejecutar nuevo código sin necesidad de alterar su implementación. Solo bastará con generar una nueva subclase que herede de **Producto** y enviarla a la función. La propia implementación que posee el método **CalculaDescuento** se encargará de hacer lo que posee implementado. Recodemos, que poder extender la funcionalidad de un programa sin tener que modificar las líneas de código que ya tenemos programadas y funcionan (**función CalcularDescuento**), es un atributo muy deseable en el desarrollo de software.

5. Agregación

La **agregación** es una relación entre clases que posee su sustento en la Jerarquía Todo-Parte que plantea el modelo orientado a objetos. Este tipo de relación permite que una clase contenedora que representa al todo, agregue una o más clases que representan las partes. Esto hace posible concretar estructuras complejas agrupando estructuras sencillas.

Este tipo de relación se define a nivel de clases pero se manifiesta a nivel de objetos. Al momento que generar una relación de agregación se puede dar de dos tipos. A la primera la denominamos simplemente **agregación** y a la segunda **composición**.

La diferencia entre ambas es que en la **agregación** la clase contenedora o que agrega no determina el ciclo de vida de lo agregado, mientras que en la **composición** sí.

Cuando decimos que la clase contenedora determina el ciclo de vida de lo contenido, establecemos que la clase contenedora crea las instancias que se necesitan y también al finalizar su existencia, finaliza la existencia de lo agregado.

En el ejemplo **Ej0023** se observan las clases **Accionista, Accion, AccionEmpresaA y AccionEmpresaB.**

```
public class Accionista
{
    1 referencia
    public Accionista(string pNombre) { this.Nombre = pNombre; }
    1 referencia
    public string Nombre { get; set; }
    private List<Accion> ListaDeAcciones = new List<Accion>();
    1 referencia
    public void AgregarAccion(Accion pAccion) {this.ListaDeAcciones.Add(pAccion); }
    1 referencia
    public void BorrarAccion(Accion pAccion) {this.ListaDeAcciones.Remove(pAccion);}
    6 referencias
    public List<Accion> VerAcciones() {return this.ListaDeAcciones; }
}
```

EJ0023

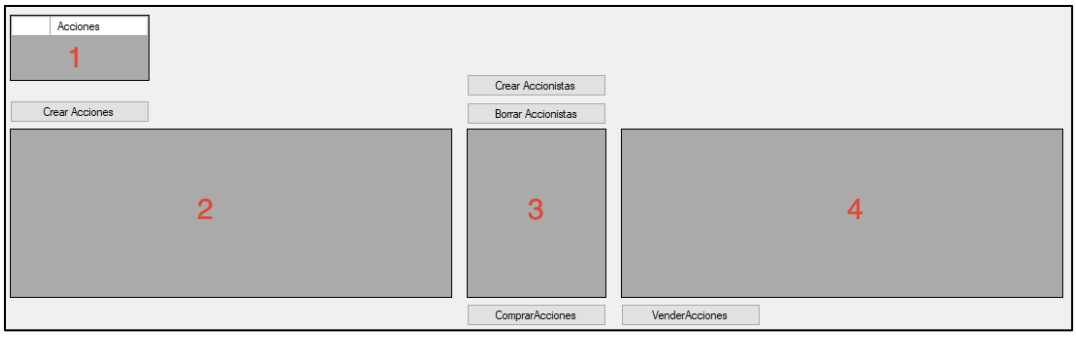
En este ejemplo un **Accionista** agrega muchas acciones. Esto lo puede realizar debido a que la clase accionista implemente una lista de acciones.

```
abstract public class Accion
    2 referencias
{   public Accion (int pCantidad,string pIdentificador,string pDescripcion,decimal pCotizacion)
    {   this.Cantidad = pCantidad;this.Identificador = pIdentificador;
        this.Descripcion = pDescripcion;this.Cotizacion = pCotizacion;
    }
    1 referencia
    public int Cantidad { get; set; }
    1 referencia
    public string Identificador { get; set; }
    1 referencia
    public string Descripcion { get; set; }
    1 referencia
    public decimal Cotizacion { get; set; }
}
2 referencias
public class AccionEmpresaA : Accion
    1 referencia
{ public AccionEmpresaA(int pCantidad, string pIdentificador, string pDescripcion, decimal pCotizacion)
        : base(pCantidad, pIdentificador, pDescripcion, pCotizacion) { }
}
2 referencias
public class AccionEmpresaB : Accion
    1 referencia
{ public AccionEmpresaB(int pCantidad, string pIdentificador, string pDescripcion, decimal pCotizacion)
      : base(pCantidad, pIdentificador, pDescripcion, pCotizacion) { }
}
```

Ej0023

La instanciación de los accionistas y de las acciones se producen en momentos distintos. Esto corrobora que los ciclos de vida de ambos son distintos.

Un accionista puede ser creado o borrado, aún cuando este deja de existir, las acciones que tenía siguen existiendo. Claramente en el ejemplo **Ej0023** se aplica el concepto tradicional de **agregación.**



Ej0023

La grilla 1 posee los dos tipos de acciones que se pueden crear, la grilla 2 muestra las acciones creadas, la grilla 3 expone los accionistas creados y la grilla 4 muestra las acciones del accionista seleccionado en la grilla 3.

A continuación se exponen los fragmentos de códigos utilizados para el ejemplo **Ej0023.** El primero muestra como se configuran las grillas.

```
public Form1()...
List<Accionista> ListaAccionista = new List<Accionista>();
List<Accion> ListaAccion = new List<Accion>();
1 referencia
private void Form1_Load(object sender, EventArgs e)
{
    #region "Configuración de la Grilla"

    this.dataGridView1.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    this.dataGridView2.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    this.dataGridView3.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    this.dataGridView4.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    this.dataGridView1.Rows.Add(2);
    this.dataGridView1.Columns[0].Width = 100;
    this.dataGridView1.Rows[0].Cells[0].Value="Acción Empresa A";
    this.dataGridView1.Rows[1].Cells[0].Value = "Acción Empresa B";
    #endregion
}
```

Ej0023

El código que se expone a continuación es el que permite crear las acciones.

```
public Form1()...
List<Accionista> ListaAccionista = new List<Accionista>();
List<Accion> ListaAccion = new List<Accion>();
1 referencia
private void Form1_Load(object sender, EventArgs e)
{
    #region "Configuración de la Grilla"

    this.dataGridView1.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    this.dataGridView2.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    this.dataGridView3.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    this.dataGridView4.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    this.dataGridView1.Rows.Add(2);
    this.dataGridView1.Columns[0].Width = 100;
    this.dataGridView1.Rows[0].Cells[0].Value="Acción Empresa A";
    this.dataGridView1.Rows[1].Cells[0].Value = "Acción Empresa B";
    #endregion
}
```

Ej0023

A continuación el código que permite crear **Accionistas.**

```
private void button4_Click(object sender, EventArgs e)
{
    this.ListaAccionista.Add(new Accionista(Interaction.InputBox("Nombre: ")));
    this.dataGridView3.DataSource = null; this.dataGridView3.DataSource = this.ListaAccionista;
}
```

Ej0023

El siguiente fragmento de código permite que un accionista pueda adquirir acciones.

```
private void button1_Click(object sender, EventArgs e)
{
    try
    {
        // Se agrega la acción comprada al accionista que la compro
        ((Accionista)this.dataGridView3.SelectedRows[0].DataBoundItem).AgregarAccion(
       (Accion)(this.dataGridView2.SelectedRows[0].DataBoundItem));
        // Se retira la accion de la lista de acciones disponibles
        this.ListaAccion.Remove((Accion)(this.dataGridView2.SelectedRows[0].DataBoundItem));
        // Se refresca la lista que muestra las acciones disponibles
        this.dataGridView2.DataSource = null; this.dataGridView2.DataSource = this.ListaAccion;
        // Se muestra las acciones del Accionista seleccionado
        this.dataGridView4.DataSource = null; this.dataGridView4.DataSource =
        ((Accionista)this.dataGridView3.SelectedRows[0].DataBoundItem).VerAcciones();
    }
    catch (Exception)
    { }
}
```

Ej0023

A continuación el código que permite que un accionista venda sus acciones.

```
private void button2_Click(object sender, EventArgs e)
{
    try
    {
        // Se selecciona la acción de la grilla de acciones que ya fueron compradas
        Accion A = (Accion)(this.dataGridView4.SelectedRows[0].DataBoundItem);
        // Se saca la accion de la lista de acciones del accionista
        ((Accionista)(this.dataGridView3.SelectedRows[0].DataBoundItem)).BorrarAccion(A);
        // Se agrega la accion a la lista de acciones disponibles
        this.ListaAccion.Add(A);
        // Se refresca la grilla de acciones disponibles
        this.dataGridView2.DataSource = null; this.dataGridView2.DataSource = this.ListaAccion;
        // Se refresca la grilla de acciones del accionista seleccionado
        this.dataGridView4.DataSource = null; this.dataGridView4.DataSource =
       ((Accionista)this.dataGridView3.SelectedRows[0].DataBoundItem).VerAcciones();

    }
    catch (Exception)
    { }
}
```

Ej0023

El siguiente código es el que permite borrar a un accionista.

```
private void button5_Click(object sender, EventArgs e)
{
    try
    {
        // si posee acciones se pasan como disponibles
        Accionista ACC = (Accionista)(this.dataGridView3.SelectedRows[0].DataBoundItem);
        if(ACC.VerAcciones().Count>0)
        {
            this.ListaAccion.AddRange(ACC.VerAcciones());
        }
        // Se borra el accionista
        this.ListaAccionista.Remove(ACC);
        // Se refresca la grilla de acciones disponibles
        this.dataGridView2.DataSource = null; this.dataGridView2.DataSource = this.ListaAccion;
        // Se refresca la grilla de accionistas
        this.dataGridView3.DataSource = null; this.dataGridView3.DataSource = this.ListaAccionista;
        // Se refresca la grilla de acciones del accionista seleccionado
        this.dataGridView4.DataSource = null; this.dataGridView4.DataSource =
            ((Accionista)this.dataGridView3.SelectedRows[0].DataBoundItem).VerAcciones();
    }
    catch (Exception)
    {}
}
```

Ej0023

Finalmente el código que permite que cuando se selecciona un accionista, se actualice la grilla que contiene las acciones que están en su posesión.

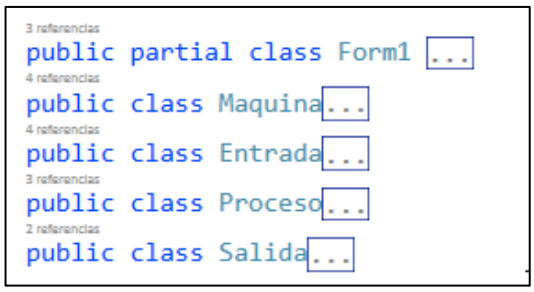
```
private void dataGridView3_CellEnter(object sender, DataGridViewCellEventArgs e)
{
    try
    {
        // Se muestra las acciones del Accionista seleccionado
        this.dataGridView4.DataSource = null; this.dataGridView4.DataSource =
        ((Accionista)this.dataGridView3.SelectedRows[0].DataBoundItem).VerAcciones();
    }
    catch (Exception)
    {}
}
```

Ej0023

La **composición,** como se expresó anteriormente, se manifiesta en relaciones donde los ciclos de vida del que agrega y los agregados estan relacionados. Esto significa que cuando se crea al que agrega, se crean los agregados. Cuando programamos, para poder hacer uso de esta posibilidad se debe acudir al uso de los constructores y los finalizadores.

A modo de ejemplo, supongamos que poseemos una máquina que está compuesta por tres elementos constitutivos y propios. El primer elemento sirve para introducirle el input o cantidad de materiales que le suministramos, se denominará *Entrada*. El segundo, es el que realiza el proceso o sea que produce los productos terminados que comercializamos, llevará ese mismo nombre *Proceso*. Finalmente la tercera, nos informa cuanto producto terminado obtuvimos en base al input suministrado y cuanto material sobró si este fuera el caso, se denominará *Salida*.

En particular consideremos que nuestra máquina produce bronce y para lograrlo se mezcla 90% de cobre y 10% de estaño. En el ejemplo **Ej0024** vemos implementado el ejemplo.



Ej0024

El ejemplo utiliza cinco clases, **Máquina** quien compone (agrega condicionando los ciclos de vida de los agregados). **Entrada**, **Proceso**, **Salida** que representan las clases agregadas. Finalmente la clase **Form1** que es la que instancia a la máquina, la utiliza y define el fin de su existencia.

```
public partial class Form1 : Form
{
    1 referencia
    public Form1()
    {InitializeComponent();}
    Maquina M =null;
    1 referencia
    private void button1_Click(object sender, EventArgs e)
    {M = new Maquina(); }
    1 referencia
    private void button2_Click(object sender, EventArgs e)
    { M.Dispose();}
    1 referencia
    private void button3_Click(object sender, EventArgs e)
    {M.Procesar();}
}
```

Ej0024

La clase **Maquina** posee la siguiente implementación:

```
public Maquina()
{
    //Constructor donde se instancian los agregados
    E = new Entrada(decimal.Parse(Interaction.InputBox("Cantidad ingresada de cobre en Kg: ")),
                    decimal.Parse(Interaction.InputBox("Cantidad ingresada de estaño en Kg: ")));
    P = new Proceso();
    S = new Salida();
    MessageBox.Show("Se han creado los objetos Entrada, Proceso y Salida !!!");
}
1 referencia
public void Procesar()
{
    P.ProducirBronce(E);
    S.VerResultados(P);
}
1 referencia
public void Dispose()
{
    //Finalizador que puede ser llamado por el usuario. El mismo rompe la composición
    this.E = null; this.P = null; this.S = null;
    MessageBox.Show("Dispose: Se han destruido los objetos Entrada, Proceso y Salida !!!");
    this.SeUsoDispose = true;
}
0 referencias
~Maquina()
{
    //Finalizador que es llamdo por el Garbage Colector. El mismo rompe la composición en caso que no se
    //haya roto por el llamado al Dispose
    if (!this.SeUsoDispose)
    {
        this.E = null; this.P = null; this.S = null;
        MessageBox.Show("Garbage Collector: Se han destruido los objetos Entrada, Proceso y Salida !!!");
    }
}
```

Ej0024

En el contructor de la clase **Maquina**, se observa la instanciación de los tres objetos que se necesitan para trabajar. Un objeto **Entrada,** un objeto **Proceso** y un objeto **Salida.** Luego el método **Procesar** de **Maquina** es utilizado el objeto **Proceso,** al cual se le solicita **ProducirBronce** y se le envía como parámetro el objeto **Entrada.** Finalmente el mismo método (**Procesar**), utiliza el objeto **Salida** y le solicita **VerResultados**, enviándole como parámetro el objeto **Proceso.**

También se puede observar en la clase **Maquina** el método **Dispose** y el finalizador **~Maquina**. En ellos es donde se predispone la situación para que se produzca la finalizción del ciclo de vida de los tres objetos: **Entrada, Proceso** y **Salida**, haciendo que las variables que lo apuntaban: **E, P** y **S** respectivamente, apunten a **null.** Esto produce que los objetos queden desapuntados, por lo tanto se transforman en basura dentro de la memoria, para que el garbage collector los elimine, generando la finalización de sus ciclos de vida.

Queda explísita la creación de los objetos agregados en el constructor de la clase **Maquina,** quien genera la composición, y en ella misma la finalización de los ciclos de vida de los agregados, **Entrada, Proceso y Salida,** en los finalizadores de la clase que les dio origen.

A continuación los fragmentos de código de las clases **Entrada**, **Proceso** y **Salida.**

```
public class Entrada
{
    private decimal VCantidadCobreKg = 0;
    private decimal VCantidadEstanoKg = 0;
    1 referencia
    public Entrada(decimal pCantidadCobreKg, decimal pCantidadEstanoKg)
            { this.CantidadCobreKg = pCantidadCobreKg; this.CantidadEstanoKg = pCantidadEstanoKg; }
    3 referencias
    public decimal CantidadCobreKg
    {
        get { return VCantidadCobreKg; }
        set {
            if (value < 1) { MessageBox.Show("La cantidad debe ser mayor a 1 Kg"); }
            else { this.VCantidadCobreKg = value; }
            }
    }
    3 referencias
    public decimal CantidadEstanoKg
    {
        get { return VCantidadEstanoKg; }
        set
        {
            if (value < 1) { MessageBox.Show("La cantidad debe ser mayor a 1 Kg"); }
            else { this.VCantidadEstanoKg = value; }
        }
    }
}
3 referencias
```

Ej0024

```
public class Proceso
{
    decimal VQCobre = 0;
    decimal VQEstano = 0;
    decimal VQBronce = 0;
    1 referencia
    public decimal SobranteCobre
    { get { return VQCobre; } }
    1 referencia
    public decimal SobranteEstano
    { get { return VQEstano; } }
    1 referencia
    public decimal BronceProducido
    { get { return VQBronce; } }
    1 referencia
    public void ProducirBronce(Entrada pEntrada)
    {
        if (pEntrada.CantidadCobreKg < 1 || pEntrada.CantidadEstanoKg < 1)
        { MessageBox.Show("Alguna o ambas cantidades de los metales de ingreso es menor a 1 !!!"); }
        else
        {
            this.VQCobre = pEntrada.CantidadCobreKg;
            this.VQEstano = pEntrada.CantidadEstanoKg;
            while (VQCobre >= 1)
            {
                if (VQEstano >= 9) { VQBronce++; VQCobre--; VQEstano -= 9; }
                else {break;}
            }
        }
    }
}
```

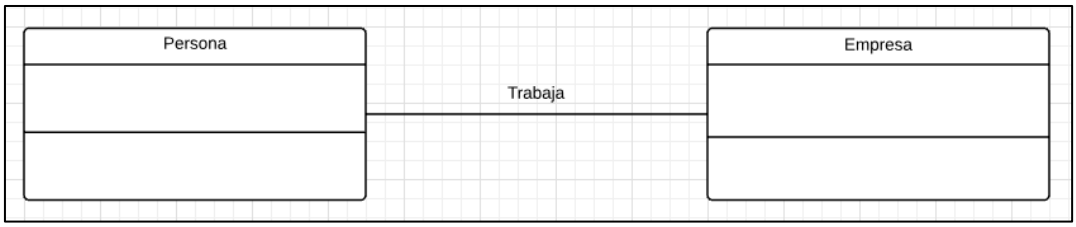
Ej0024

```
public class Salida
{
    1 referencia
    public void VerResultados(Proceso pProceso)
    {
        MessageBox.Show("El bronce producido es: " + pProceso.BronceProducido + " Kg\n" +
                        "Cobre sobrante: " + pProceso.SobranteCobre + " Kg\n" +
                        "Estaño sobrante: " + pProceso.SobranteEstano + " Kg");
    }
}
```

Ej0024

3. Asociación y Relación de Uso

La relación de **asociación** se da cuando las clases se conectan de forma conceptual. A diferencia de las anteriores relaciones, no observamos una relación todo-parte. La **asociación** surge de la propia relación con un verbo que la identifique.

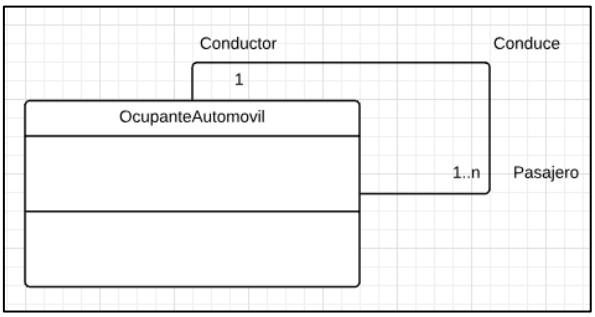


En una **asociación** se pueden establecer roles, como el rol de **empleador** y **empleado** en la relación precedente.

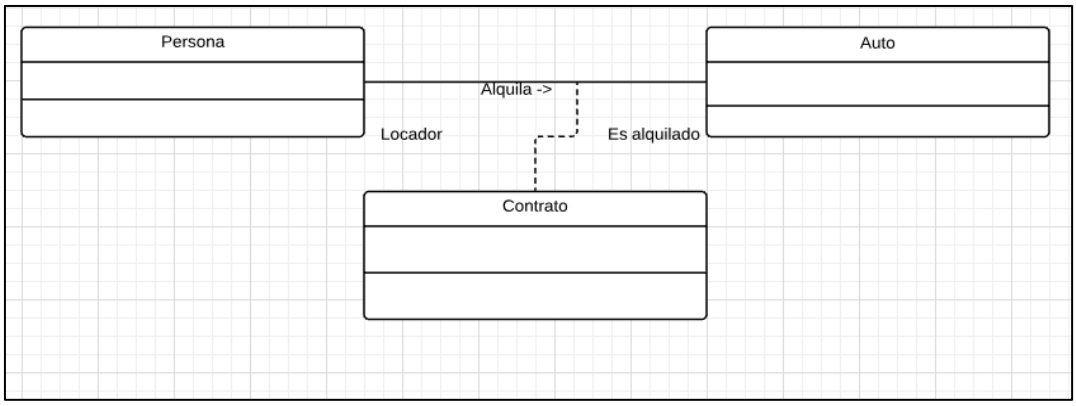
Se puede establecer una asociación bidireccional como se puede observar a continuación:

También se pueden producir asociaciones reflexivas o unarias. Esto se da cuando existen casos en los que una clase se puede

relacionar consigo misma. Cuando una clase se puede relacionar consigo misma es porque puede adquirir diferentes roles.



Existen oportunidades que en las asociaciones nos encontramos con que necesitan atributos y/o métodos específicos. Esto se sustenta en la necesidad de manejar de forma idónea y más eficiente los elementos que surgen de dicha relación. A estas clases las denominamos **clases asociativas**. En el siguiente ejemplo la nueva clase **Contrato** contiene atributos y métodos específicos de la relación entre el **Locatario** y el **Locador**.



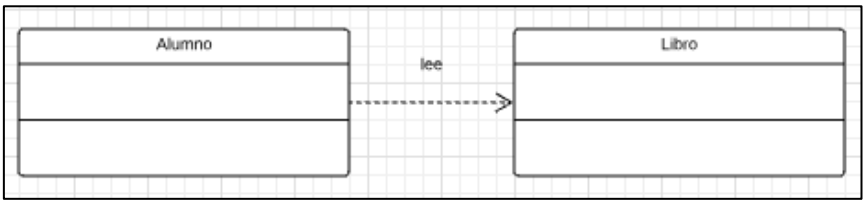
En algunas oportunidades es posible que se generen dudas para determinar si entre dos clases existe una relación de asociación o una de agregación.

Para facilitar esta decisión se pueden revisar algunos aspectos que facilitarán la tarea. En primer lugar la agregación determina relaciones de todo-parte, por lo cual siempre tendremos elementos que son “partes” del “todo”. En general cuando se existe una asociación se detecta una conexión del tipo “usa un” o “tiene un”.

En las relaciones de asociación se permite el conocimiento de ambas clases, es decir, puede existir la bidireccionalidad, sin embargo las relaciones de agregación son unidireccionales. El "todo" conoce a las "partes" pero no al revés. Otro aspecto a considerar es que la cardinalidad de la agregación es de 1..1 (uno a uno) o 1..n (uno a muchos), mientras que en la asociación puede ser de 1..1 (uno a uno), 1..n (uno a muchos) y n..n (muchos a muchos).

Para finalizar este punto definimos que es una **relación de uso**. Una **relación de uso** es una asociación refinada, donde se establece quien va a hacer uso de quien.

Por ejemplo si tenemos la clase **Alumno** y la clase **Libro**, y entre ellas existe una relación donde **Alumno** retira un libro para leer, entre ellas existe una **relación de uso**, o sea, es básicamente una asociación, pero claramente **Alumno** usa **Libro** para leerlo.



En el ejemplo **Ej0025** se puede observar como el método **Lee** recibe una referencia al libro que tiene que leer. Los siguientes fragmentos de código ejemplifican lo dicho.

```
public partial class Form1 : Form
{
    1 referencia
    public Form1() {InitializeComponent(); }
    1 referencia
    private void Form1_Load(object sender, EventArgs e) { }
    1 referencia
    private void button1_Click(object sender, EventArgs e)
    {
        Alumno A = new Alumno(Interaction.InputBox("Ingrese el nombre del alumno: "));
        Libro L = new Libro(Interaction.InputBox("Ingrese el título del libro: "),
                            Interaction.InputBox("Ingrese el contenido del libro: "));
        MessageBox.Show("El Alumno " + A.Nombre + " ha leido el libro '" + L.Titulo + "'" +
            "\n\nCuyo contenido es: " + A.Lee(L));
    }
}
```

Ej0025

```
public class Alumno
{
    private string Vnombre;
    1 referencia
    public Alumno(string pNombre) {this.Vnombre = pNombre;}
    1 referencia
    public string Nombre { get { return this.Vnombre; } }
    1 referencia
    public string Lee(Libro pLibro) { return pLibro.Contenido;}
}
4 referencias
public class Libro
{
    private string Vcontenido;
    private string Vtitulo;
    1 referencia
    public Libro(string pTitulo, string pContenido)
    {this.Vcontenido = pContenido; this.Vtitulo = pTitulo;}
    1 referencia
    public string Contenido { get { return this.Vcontenido; } }
    1 referencia
    public string Titulo { get { return this.Vtitulo; } }
}
```

Ej0025

EL CÓDIGO QUE SE UTILIZA EN LAS EXPLICACIONES LO PUEDE BAJAR DEL **MÓDULO RECURSOS Y BIBLIOGRAFÍA**.

Actividades asincrónicas

Guía de preguntas de repaso conceptual

1. ¿Qué cosas se pueden heredar?
2. ¿Cómo y para qué se puede aprovechar en la práctica el polimorfismo?
3. ¿Cómo y para qué se utiliza la clase derived?
4. ¿Qué representa this?
5. ¿Qué clase representa a la clase base?
6. ¿Para qué se usa una clase abstracta?
7. ¿Para qué se usa una clase sellada?
8. ¿Qué es la sobrescritura?
9. ¿Qué elementos se pueden sobrescribir?
10. ¿A qué se denomina sombreado de métodos?

Guía de ejercicios

1. Desarrollar un programa donde se observe claramente el uso de los miembros de base en un entorno de herencia. Enfatice las particularidades al usarlo en los constructores y finalizadores. Demuestre en el mismo programa el uso de this. Establezca las diferencias y en qué caso se justifica utilizar cada uno.
2. Desarrollar un programa que posea al menos una clase abstracta, un método virtual, una clase sellada y una clase anidada.