UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Centro de e-Learning

FUNDAMENTOS DE LA PROGRAMACIÓN

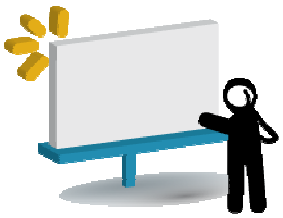
Centro de Elearning - FRBA - UTN

Centro de e-Learning SCEU UTN - BA.
Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Centro de e-Learning - FRBA - UTN

MÓDULO 3 - UNIDAD 11

Conceptos Avanzados



Presentación:

Con esta Unidad, que cierra los aspectos netamente orientados al paradigma de la POO, se analizan los conceptos más avanzados del mismo.

Así mismo se analiza otro de los mecanismos más poderosos del paradigma: el Polimorfismo, otro de los pilas de la POO.

Otros de los temas que serán analizados y que tiene un correlato directo con las prácticas de programación y los lenguajes de programación actuales son las clases abstractas y las interfaces, ambos temas muy importantes de ser adquiridos.

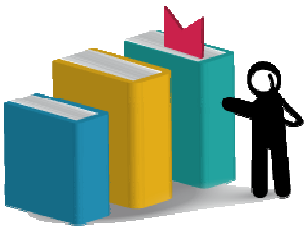
Centro de Elearning - FRBA - UTN

Objetivos:

Que los participantes:

- Incorporen los conceptos de interfaces y clases abstractas.
- Incorporen los conceptos de casting, o transformación de tipos de datos.
- Comprendan el mecanismo del Polimorfismo.
- Comprendan el concepto de clases internas.

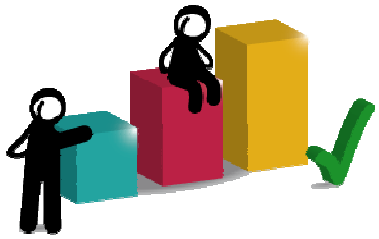
Centro de Elearning - FRBA - UTN



Bloques temáticos:

1. Interfaces
2. Clases abstractas
3. Clases internas
4. Casteo ("casting")
5. Principios de la OO: Polimorfismo

Centro de Elearning - FRBA - UTN



Consignas para el aprendizaje colaborativo

En esta Unidad los participantes se encontrarán con diferentes tipos de actividades que, en el marco de los fundamentos del MEC*, los referenciarán a tres comunidades de aprendizaje, que pondremos en funcionamiento en esta instancia de formación, a los efectos de aprovecharlas pedagógicamente:

- Los foros proactivos asociados a cada una de las unidades.
- La Web 2.0.
- Los contextos de desempeño de los participantes.

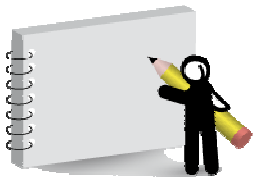
Es importante que todos los participantes realicen algunas de las actividades sugeridas y compartan en los foros los resultados obtenidos.

Además, también se propondrán reflexiones, notas especiales y vinculaciones a bibliografía y sitios web.

El carácter constructivista y colaborativo del MEC nos exige que todas las actividades realizadas por los participantes sean compartidas en los foros.

* El MEC es el modelo de E-learning colaborativo de nuestro Centro.

Centro de E-learning - FRBA - UTN



Tomen nota

Las actividades son opcionales y pueden realizarse en forma individual, pero siempre es deseable que se las realice en equipo, con la finalidad de estimular y favorecer el trabajo colaborativo y el aprendizaje entre pares. Tenga en cuenta que, si bien las actividades son opcionales, su realización es de vital importancia para el logro de los objetivos de aprendizaje de esta instancia de formación. Si su tiempo no le permite realizar todas las actividades, por lo menos realice alguna, es fundamental que lo haga. Si cada uno de los participantes realiza alguna, el foro, que es una instancia clave en este tipo de cursos, tendrá una actividad muy enriquecedora.

Asimismo, también tengan en cuenta cuando trabajen en la Web, que en ella hay de todo, cosas excelentes, muy buenas, buenas, regulares, malas y muy malas. Por eso, es necesario aplicar filtros críticos para que las investigaciones y búsquedas se encaminen a la excelencia. Si tienen dudas con alguno de los datos recolectados, no dejen de consultar al profesor-tutor. También aprovechen en el foro proactivo las opiniones de sus compañeros de curso y colegas.

Centro de Elearning FRBA - UTN

1. Interfaces

En la mayoría de los lenguajes OO, una clase es escrita o "existe" dentro de un archivo físico. En Java, estos archivos se llaman, por ejemplo, "Persona.java". En C++, podría ser "Persona.cpp" ("cpp" = "c plus plus", la forma inglesa de llamar al lenguaje, siendo la traducción al español "c más más").

Por lo que podemos decir que una clase, en realidad, es un archivo, si lo llevamos al "plano físico".

Aquí hacemos una salvedad, al mencionar que algunos lenguajes implementan el concepto de "inner classes", o clases internas, en la cual es posible "anidar" una clase dentro de otra; concepto que desarrollaremos más adelante en esta misma Unidad.

Hecha la salvedad, podemos decir que mayormente una clase se corresponde a un archivo.

También nos podemos encontrar algunos otros elementos, como las interfaces o las clases abstractas (conceptos que revisaremos a continuación), que suelen encontrarse también en archivos con la misma extensión que una clase, por lo que será necesario revisar el contenido y comprobar si se trata de una clase o interface antes de poder usarla o invocarla.

Como primera aclaración sobre las interfaces, al menos dentro de los conceptos de la POO, aclaramos las confusiones que trae este concepto, al diferenciarlas de las "interfaces de usuario", haciendo referencia a un frontend o capa de presentación de una aplicación, o a las "interfaces entre sistemas", entiendo por estas como un canal por el cual se comunican dos o más aplicaciones.

Centro de e-Learning - FRBA - UTN

Hablando de POO, una interface (o "interfaz", las llamaremos de una u otra forma) se define como un conjunto de atributos y/o métodos sin implementar, o sea, con los métodos "vacíos", presentando solamente la firma de los mismos.

Esto significa que una interface no va a contener comportamiento, pero sí va a decirle a las clases que la usan cómo deberán comportarse.

Cuando una clase "usa" una interface, se dice que la "**implementa**". O sea, la clase termina siguiendo las reglas que le impone la interface.

La **utilidad de las interfaces radica en que a través de ellas cobra valor el principio de la ocultación**, siendo este uno de los pilares de la POO. Este principio sostiene que cada objeto debe estar aislado del exterior y que a su vez expone al exterior una interface que especifica cómo pueden interactuar otros objetos con él. **El aislamiento que propone este principio permite resguardar los atributos y métodos de un objeto contra su modificación por cualquier medio que no tenga derecho a acceder a ellos**, logrando de esta forma mantener el estado interno de un objeto sin alteraciones o interacciones inesperadas.

Repasando, al ser este un punto que suele prestarse a confusión, podemos decir que una interface:

- Tiene una "forma" parecida a una clase
- Puede tener atributos
- Puede tener métodos, pero sólo sus firmas, sin implementar
- Se dice que una clase que usa una interface, la "implementa"
- Una interface puede ser implementada por un número indefinido de clases
- Una clase puede implementar un número indefinido de interfaces

Centro de e-Learning - FRBA - UTN

- Una clase que implementa una interface puede hacer uso de los atributos definidos en la misma y **DEBE implementar TODOS los métodos que tenga**
- Su uso se debe principalmente al principio de ocultación de la POO
- Permite definir cómo deberán comportarse el grupo de clases que la implementen

En el siguiente ejemplo, veremos cómo una clase implementa una interface. Su uso se vuelve importante cuando designamos que un conjunto de clases tenga que comportarse de una determinada manera, ya que no sería muy común que una interface sea implementada únicamente por una clase (aunque puede ocurrir).

Veremos que en el ejemplo se introduce **una nueva palabra reservada, "interface" e "implementa"** a nuestro pseudocódigo.

```
interface publica Animal
    metodo publico respiracion()
fin interface

clase publica Hombre implementa Animal
    metodo publico respiracion()
        mostrar: "Respiración pulmonar"
    fin metodo
fin clase

clase publica Pez implementa Animal
    metodo publico respiracion()
        mostrar: "Respiración branquial"
    fin metodo
fin clase
```

De forma análoga al uso de la Herencia podemos decir que, según esta estructura, un “Hombre” ES del tipo “Animal” y que un “Pez” es también del tipo “Animal”. No se puede hacer la relación inversa (un “Animal” no es un “Hombre”).

También podemos ver que la interface “Animal” se encuentra la firma del método “respiración”. Recordar que el concepto de “firma” lo analizamos en la Unidad anterior. En ambas clases que implementan la interface, se puede ver como cada una la implementa de forma particular, según las necesidades del caso.

Centro de Elearning - FRBA - UTN

2. Clases abstractas

Las clases abstractas no están implementadas en todos los lenguajes OO, aunque es interesante nombrarlas ya que están presentes en la mayoría de los lenguajes más difundidos en la actualidad.

Al igual que las interfaces, se presentan en archivos físicos similares a las clases. Para identificarlas, **incorporaremos la palabra reservada "abstracta"** en la declaración de la clase.

Pueden presentar métodos sin implementar, colocando solamente la firma del método (conocidos como métodos abstractos), aunque a diferencia de las interfaces, estas clases sí permiten implementar sus métodos.

Es condición para que una clase sea abstracta que al menos uno de sus métodos sea abstracto, ya que esto es evaluado por el compilador, a modo de validación, además de ser "una regla" del paradigma OO.

Otra similitud que guardan con las interfaces es que tampoco pueden instanciarse. Dicho de otra forma, **no es posible crear un objeto en base a una clase abstracta**, justamente por su naturaleza de abstracción que implica un nivel demasiado alto o generalista.

Por esto**, la única forma de utilizar una clase abstracta es a través de la herencia**, con otra clase que la extienda (que herede de ella). De esta forma, los métodos implementados pasan a ser "utilizables" por la clase hija cuando se la instancia, debiendo implementar aquellos métodos no implementados en la misma clase hija.

Centro de E-Learning FRBA - UTN

Así como la interface dice cuál va a ser el comportamiento de una clase, la clase abstracta aclara también el detalle de ese comportamiento.

En el siguiente ejemplo, podemos ver como dos clases “Gerente” y “Supervisor”, que heredan de la clase “Empleado”, siendo ésta abstracta, tienen todos los atributos y métodos de la clase padre, pero cada una de estas debe implementar un método en particular llamado “calculaSueldoConBeneficios”, ya que la forma de cálculo de los beneficios varía entre los tipos de empleados.

```
clase publica abstracta Empleado
    //atributos de la entidad
    protegido Float sueldoBase nuevaInstancia Float()
    protegido Float sueldoConBeneficios nuevaInstancia Float()

    //cálculos de negocio
    metodo publico Float sueldoBase()
        sueldoBase = 1000
        retornar: sueldoBase
    fin método

    metodo publico abstracto Float calculaSueldoConBeneficios()
fin clase
```

Centro de e-Learning FRBA - UTN

```
clase publica Supervisor heredaDe Empleado
    metodo publico Float calculaSueldoConBeneficios()
        sueldoConBeneficios = sueldoBase * 1,2
        retornar: sueldoConBeneficios
    fin metodo
fin clase

clase publica Gerente heredaDe Empleado
    metodo publico Float calculaSueldoConBeneficios()
        sueldoConBeneficios = sueldoBase * 1,3
        retornar: sueldoConBeneficios
    fin metodo
fin clase
```

Tanto la clase “Supervisor” como “Gerente” heredan la clase abstracta “Empleado”. Esta clase es abstracta, ya que tiene la firma de un método abstracto llamado “sueldoConBeneficios()”.

Las clases hijas deben implementar el método abstracto definido en la clase padre. Al hacerlo, cada una deberá implementar el método con las particularidades que corresponda. En este caso, cada clase hija hace un cálculo diferente.

En el que caso de encontrarnos con una clase abstracta que tiene TODOS sus métodos abstractos sin implementar, deberíamos preguntarnos si esa clase abstracta no debería ser una interfaz.

3. Clases internas

Las “inner classes” son un mecanismo que proveen algunos lenguajes OO que **permite definir una clase dentro de otra**; de forma similar a la que una clase puede tener métodos y atributos, a través de este concepto también podría contener otra clase.

Su uso resulta de utilidad cuando se tienen que crear clases que tienen una relación conceptual muy cercana entre sí, en las que es necesario compartir los métodos y atributos, incluso aquellos definidos como privados.

La clase interna es tomada como un miembro más de la clase externa (la que la contiene), por lo que se puede aplicar sobre la interna todos los modificadores de acceso en forma similar a como se hace con atributos y métodos.

El uso de clases internas es una práctica que tiende a quedar en desuso, dado que se le critica el hecho de agregar complejidad y confusión al código de la clase y de aquellas que las usan.

Por este motivo, en lo que respecta a este curso, nos quedaremos con el concepto y la definición, pero no profundizaremos en la práctica.

Centro de Elearning - FRBA - UTN

4. Casteo ("casting")

Para este concepto mantendremos su nombre en inglés, dado que es el más utilizado y difundido, sumado a que tampoco existe una traducción ampliamente aceptada sobre el término, siendo la más difundida “moldeado”. Incluso, más que este término, se encuentra difundido el de “casteo”, como forma castellanizada de definir esta técnica, cuya función se centra básicamente en **forzar la conversión de un valor de un atributo de un tipo de dato a otro**.

Para esto se utiliza el operador “cast” (que introducimos como nueva palabra reservada), cuya sintaxis es:

(tipo)expresión

De esta forma, se obtiene un tipo de dato a partir de otro distinto. Un ejemplo de uso podría ser:

```
clase publica EjemploCast
    privado Integer numeroEntero nuevaInstancia Integer()
    privado Float numeroConComa nuevaInstancia Float()

    metodo publico EjemploCast ()
        numeroConComa = 3,14
        numeroEntero = (Integer) numeroConComa
        mostrar: "El número pasado a entero es "
        mostrar: numeroEntero
    fin metodo
fin clase
```

Centro de e-Learning UTN

De esta forma, la salida de constructor de esta clase va a ser:

"El número pasado a entero es 3"

Ya que se trasformó un número con coma a otro que no soporta coma, pero que tiene cierta **coherencia conceptual (ambos son números)**. Esto significa que no tendría sentido (de hecho, produciría un error de compilación) si se quiere pasar, por ejemplo, una fecha a número con coma o un texto a número. No así, pasar una fecha a texto, práctica que suele ser utilizada para permitir la manipulación de la fecha.

El hecho de que la conversión cambie el número, se debe a que es posible que exista una pérdida de significancia al convertir datos de un tipo a otro. Esto es algo que siempre se deberá tener en cuenta cuando se realice una operación de *casting*.

Vale aclarar que el *casting* no es algo que se restringe en la práctica solamente a los objetos, ya que en algunos lenguajes de programación es posible realizar conversiones de tipo de datos primitivos.

Centro de Elearning - FRBA - UTN

5. Principios de la OO: Polimorfismo

Este es uno de los principios de la OO que más confusiones suele traer. Aunque, a su vez, es una herramienta poderosa para lograr un código claro y robusto, a la vez que se aprovechan al máximo las posibilidades que brindan la OO.

El Polimorfismo, literalmente, como su nombre lo indica, permite que un objeto tome muchas formas, a la vez que mantiene el comportamiento. Esto significa que el polimorfismo permite hacer referencia a métodos o atributos de objetos que cambian en tiempo de ejecución.

Su **uso puede darse a través de la herencia o a través del uso de interfaces**. A continuación veremos un ejemplo combinando ambos, herencia e interfaces:

```
interface publica Resultado heredaDe IntInteger
    //vamos a suponer que la clase Integer implementa tambien IntInteger
fin interface
```

```
clase publica abstracta Operacion
    metodo publico abstracto realizarOperacion(Integer a, Integer b)
    metodo publico abstracto Integer obtenerResultado()
fin clase
```

Centro de e-Learning - FRBA - UTN

```
clase publica Suma heredaDe Operacion
    privado Integer resultado nuevaInstancia Integer()

    metodo publico Suma(Integer a, Integer b)
        realizarOperacion(a, b)
    fin metodo

    metodo publico Integer obtenerResultado()
        retornar: resultado
    fin metodo

    metodo publico realizarOperacion(Integer a, Integer b)
        resultado = a + b
    fin metodo
fin clase

clase publica Multiplicacion heredaDe Operacion
    privado Integer resultado nuevaInstancia Integer()

    metodo publico Multiplicacion(Integer a, Integer b)
        realizarOperacion(a, b)
    fin metodo

    metodo publico Integer obtenerResultado()
        retornar: resultado
    fin metodo

    metodo publico realizarOperacion(Integer a, Integer b)
        resultado = a * b
    fin metodo
fin clase

clase publica HagoOperacion
    metodo publico HagoOperacion ()
        privado Operacion operacion1 nuevaInstancia Suma(1,2)
        privado Operacion operacion2 nuevaInstancia Multiplicacion(1,2)

        Resultado resultado1 = operacion1.obtenerResultado()
        Resultado resultado2 = operacion2.obtenerResultado()
    fin metodo
fin clase
```

En este ejemplo tenemos:

- Interface Resultado: extiende (hereda de) la interface IntInteger, por lo que se puede decir que un “Resultado” siempre es un “Integer”, pero no el caso contrario.
- Clase Abstracta Operacion: define dos métodos abstractos. Es la “superclase” (es lo que venimos haciendo referencia como “clase padre”) que extienden/heredan las dos clases siguientes.
- Clase Suma: extiende/hereda la superclase abstracta Operación e implementa (como corresponde) sus métodos. La clase “Suma” es una “Operacion” que sólo sabe sumar dos números.
- Clase Multiplicacion: extiende/hereda la superclase abstracta Operación e implementa sus métodos. La clase “Multiplicacion” es una “Operacion” que sólo sabe multiplicar dos números.
- Clase HagoOperacion: es una clase de ejemplo en la que podemos ver como se implementa el polimorfismo:
    - Primer caso: Polimorfismo por herencia:

      privado Operacion operacion1 nuevaInstancia Suma(1,2)
      privado Operacion operacion2 nuevaInstancia Multiplicacion(1,2)

      En este caso, creamos una instancia de “Suma” y otra de “Multiplicacion”, aunque ambos objetos son del tipo “Operacion”.

      "1" y "2" son valores numéricos enteros (de ejemplo) que se le envían como parámetros al constructor sobrecargado de la clase "Suma", siendo la firma de ese constructor "metodo Suma(Integer, Integer)".

    - Segundo caso: polimorfismo por interfaces. Esta es la implementación de polimorfismo más comúnmente encontrada.

      Sabiendo que:

      operacion1.obtenerResultado()

      Devuelve un Integer, este puede ser asignado a un Integer o cualquier otra clase o interfaz que extienda Integer.

Centro de Elearning - FRBA - UTN

Resultado resultado1 = operacion1.obtenerResultado()

En este caso (poco práctico), hubiera sido análogo hacer:

Integer resultado1 = operacion1.obtenerResultado()

El polimorfismo, como se muestra en el ejemplo anterior, suele traer ventajas aplicado desde las interfaces, ya que permite crear nuevos tipos sin necesidad de tocar las clases ya existentes (imaginemos que deseamos añadir una clase “Restar”, por ejemplo), basta con recompilar todo el código que incluye los nuevos tipos añadidos. Si se hubiera recurrido a la sobrecarga durante el diseño exigiría retocar la clase anteriormente creada al añadir la nueva operación “Restar”, lo que además podría suponer revisar todo el código donde se instancia a la clase.

Suele ser confusa la diferenciación entre los conceptos de sobrecarga y polimorfismo. Las principales diferencias son:

- La sobrecarga se da siempre dentro de una sola clase, en tanto el polimorfismo se da entre clases distintas.
- Un método está sobrecargado si dentro de una clase existe más de una declaración de dicho método con el mismo nombre pero con parámetros distintos.

La sobrecarga se resuelve en tiempo de compilación utilizando los nombres de los métodos y los tipos de sus parámetros; el polimorfismo se resuelve en tiempo de ejecución del programa, esto es, mientras se ejecuta, en función al tipo al que pertenece el objeto.

¿Por qué la clase "HagoOperacion" aplica el polimorfismo?

Porque en esa clase se crean los objetos: "operacion1" y "operacion2", por lo que obtenemos un ejemplo de polimorfismo por herencia (ver la jerarquía de herencia de las clases "Operacion", "Suma" y "Multiplicacion"). En este caso "Operacion" es abstracta, por lo que no puede instanciarse (por definición de clase abstracta), por lo que la única forma es hacerlo a través de la jerarquía de las clases (herencia). Ambas clases "Suma" y "Multiplicacion" tienen el "heredaDe Operacion"

Centro de Elearning - FRBA - UTN

Pregunta frecuente: ¿Cuál es la diferencia sustancial entre Interface y Clase abstracta en cuanto a su uso?

La realidad indica que una clase abstracta SIN métodos propios (o sea, solamente con método abstractos) no difiere en gran medida de una interfaz: ninguna de las 2 puede ser instanciada y los que las quieran usar (por herencia o implementación en cada caso) deberá desarrollar (implementar) esos métodos en sus clases.

Ahora bien, usarlas de esta forma, si bien es posible, es un error conceptual. Porque estaríamos haciendo "como que" una clase abstracta es una interfaz. No tiene sentido. Es como clavar un clavo con una llave inglesa: ¿Se puede? Sí, claro que se puede. Pero si tenemos un martillo nuestro lado... Es, al menos, dudoso.

¿Cuándo usamos cada uno? La interfaz la vamos a usar cuando queramos "obligar" o "forzar" que ciertas clases implementen ciertas funcionalidades mínimas (que serán las firmas de los métodos de la interfaz). La clase abstracta va a servir para lo mismo, con la gran diferencia de que además le vamos a dar a las clases que heredan funcionalidades ya implementadas, que son los métodos "normales" (no abstractos) de la clase abstracta.

Por ejemplo: sabemos que todas las personas tienen un nombre. No quisiéramos, entonces, modelar una entidad persona que pueda no tener nombre. Entonces vamos a definir una interfaz "Persona" que va a obligar a que todas las personas concretas que tengamos (físicas o jurídicas, por ejemplo) tengan un nombre.

Pero si quisiéramos que, además, todas las personas concretas tengan el saldo de la cuenta con $100 cuando se crean, bueno, ya podríamos definirles un método que cree la cuenta con saldo de $100, y agregar el uso del nombre como método abstracto. Y así pasamos de necesitar una interfaz a tener una clase abstracta.

¿Por qué polimorfismo?

El polimorfismo no muestra su cara más útil o sus beneficios como lo hace la herencia (por ejemplo: evitar el copypaste). Pero como alternativa que nos ofrece el paradigma para resolver problemas puntuales, es importante conocer, al menos, su concepto.

Es, también, una técnica de programación presente, sobre todo, en la ESTRUCTURA de los lenguajes. Por ejemplo, en Java está muy presente en varios usos cotidianos del lenguaje.

Por ejemplo, para usar un ArrayList (es como un vector, pero en objetos y con un muchos métodos útiles), cuando queremos crear un objeto lo hacemos así:

List list = new ArrayList();

Donde:

- **"List" es una interfaz.**
- **"list" es el objeto**
- **"new =" es equivalente a nuestro "nuevaInstancia"**
- **"ArrayList()" es el constructor por defecto de la clase ArrayList**
- **";" es el indicador de fin de línea en Java**

Este es un ejemplo común y cotidiano de uso del polimorfismo por interfaces.

Centro de E-Learning - FRBA - UTN

6. Tips de uso y ejemplos

- Interfaces

Ejemplo 1:

```
clase publica ToyotaSW42Punto8TdiSrx4Wd implementa Auto
    metodo publico MotorAuto (Boolean elec)
        mostrar:" Motor: 2.8 L 4 motor en línea diésel"
        si elec = verdadero entonces
            mostrar: "Eléctrico"
        fin si
    fin metodo

    metodo publico TransmisionAuto (Integer vel)
        mostrar:" Transmisión: manual de " + vel + " velocidades"
    fin metodo
fin clase

clase publica ToyotaCorolla1Punto8XEIHybridCVT implementa Auto
    metodo publico MotorAuto (Boolean elec)
        mostrar:" Motor: 1.8 L 4 motor en línea Hybrido"
        si elec = verdadero entonces
            mostrar: "Eléctrico"
        fin si
    fin metodo

    metodo publico TransmisionAuto (Integer vel)
        mostrar:" Transmisión: Automatica tipo CVT limitada a " + vel + " velocidades"
    fin metodo
fin clase

clase publica ChevroletTrailblazerCTDI4X4PremierAT implementa Auto
    metodo publico MotorAuto (Boolean elec)
        mostrar:" Motor Diesel de 200 CV y 500 Nm de torque"
        si elec = verdadero entonces
            mostrar: "Eléctrico"
        fin si
    fin metodo

    metodo publico TransmisionAuto (Integer vel)
        mostrar:" Transmisión: Transmisión automática de " + vel + " velocidades"
    fin metodo
fin clase


interface publica Auto
    //firmas de los métodos
    metodo publico MotorAuto (Boolean)
    metodo publico TransmisionAuto (Integer)
fin interface
```

En este ejemplo, vemos al final la interfaz. Notar que los método no están implementados, sino que solo se colocan LAS FIRMAS de los métodos (la firma, recuerden, incluye los tipos de datos de los parámetros, pero no lo nombres de los parámetros: en este caso son "Boolean" en un método y "String" en el otro.

Esta interfaz es implementada por 3 clases concretas que están arriba.

Las 3 clases SI o SI deben implementar todos los métodos de la interfaz.

Ejemplo 2:

```
interface publica Persona
    metodo publico calculoEdad(Integer)
    metodo publico definirAnioActual(Integer)
fin interface
```

```
clase publica Ninio implementa Persona

    privado Integer anioActual nuevaInstancia Integer()

    metodo publico calculoEdad(Integer anioNacimiento)
        privado Integer edad nuevaInstancia Integer()
        edad = anioActual - anioNacimiento

        si edad < 13 entonces
            mostrar: "La edad del niño es " + edad
        fin si
    fin metodo

    metodo publico definirAnioActual(Integer anio)
        anioActual = anio
    fin metodo

    metodo publico Ninio()
        mostrar: "se va a crear una persona niño"
    fin metodo
fin clase
```

```
clase publica Adulto implementa Persona
    privado Integer anioActual nuevaInstancia Integer()

    metodo publico Adulto ()
        mostrar: "se va a crear una persona adulta"
    fin metodo

    metodo publico definirAnioActual(Integer anio)
        anioActual = anio
    fin metodo

    metodo publico calculoEdad(Integer anioNacimiento)
        privado Integer edad nuevaInstancia Integer()

        edad = anioActual - anioNacimiento

        si edad > 13 entonces
            mostrar: "La edad del adulto es " + edad
            en caso de edad
                caso > 50
                    mostrar: "Es un babyboomer"
                fin caso
                caso > 40
                    mostrar: "Es un generación X"
                fin caso
                caso > 25
                    mostrar: "Es un millenial"
                fin caso
                sino
                    mostrar: "Es un centenial"
            fin en caso de
        sino
            mostrar: "Es un adulto con espíritu de niño"
        fin si
    fin metodo

fin clase
```

```
clase publica CreoPersonas
    metodo publico CreoPersonas()
        privado Ninio ninioDe12 nuevaInstancia Ninio()
        ninioDe12.definirAnioActual(2020)
        ninioDe12.calculoEdad(2008)

        privado Ninio ninioDe9 nuevaInstancia Ninio()
        ninioDe9.definirAnioActual(2020)
        ninioDe9.calculoEdad(2011)

        privado Adulto adultoDe41 nuevaInstancia Adulto()
        adultoDe41.definirAnioActual(2020)
        adultoDe41.calculoEdad(1978)

        privado Adulto adultoDe28 nuevaInstancia Adulto()
        adultoDe28.definirAnioActual(2020)
        adultoDe28.calculoEdad(1992)

    fin metodo
fin clase
```

En este ejemplo vamos a crear personas sobre las cuales se va a calcular la edad. Vamos a diferenciar adultos de niños, porque de los adultos me interesa saber (además de la edad) a qué generación pertenecen. De los niños no. Por lo tanto se crean 2 clases diferentes: "Ninios" y "Adultos". Cada una va a tener un comportamiento diferentes. Son parecidas, pero no iguales. Y, de hecho, realizan diferentes acciones.

Pero como queremos que sea obligatorio el cálculo de la edad para todos los tipos de personas que existan, vamos a hacer que las clases contengan un mínimo de comportamiento que deben tener si o si. Para esto, definimos que las clases van a implementar la interfaz "Persona", que va a contener la estructura básica, el mínimo de comportamiento que toda clase que modele personas debe tener.

FRBA UTN

- Clases abstractas

```
clase publica abstracta Figura
    protegido String color nuevaInstancia String()

    metodo publico Figura(String c)
        color = c
    fin metodo

    metodo publico abstracto Float calculoArea(String)

    metodo publico String obtenerColor()
        retornar: color
    fin metodo
fin clase


clase publica Cuadrado heredaDe Figura
    privado Float lado nuevaInstancia Float()

    metodo publico Cuadrado(String colorN, Float ladoN)
        color = colorN      //modifico el atributo de la clase padre
        lado = ladoN
    fin metodo

    metodo publico Float calculoArea(String quienSoy)
        mostrar: "Hola! Soy un " + quienSoy
        retornar: lado * lado
    fin metodo
fin clase


clase publica Triangulo heredaDe Figura
    privado Float base nuevaInstancia Float()
    privado Float altura nuevaInstancia Float()

    metodo publico Triangulo(String colorN, double baseN, double alturaN)
        color = colorN      //modifico el atributo de la clase padre
        base = baseN
        altura = alturaN
    fin metodo

    metodo publico Float calculoArea(String quienSoy)
        mostrar: "Hola! Soy un " + quienSoy
        retornar: (base * altura) / 2
    fin metodo
fin clase
```

```
clase publica CrearCuadrado
    metodo publico CrearCuadrado()
        privado String color nuevaInstancia String()

        privado Float lado nuevaInstancia Float()

        privado Float base nuevaInstancia Float()
        privado Float altura nuevaInstancia Float()


        //*****   Creo un cuadrado!!
        mostrar: "Ingresar color del cuadrado:"
        ingresar: color
        mostrar: "Ingresar lado del cuadrado:"
        ingresar: lado

        privado Cuadrado c1 nuevaInstancia Cuadrado(color, lado)
        //invoco la implementación del metodo abstracto de la clase Cuadrado
        mostrar: "El área del cuadrado " + c1.calculoArea("cuadrado!!!")
        //invoco el método heredado de la clase Cuadrado
        mostrar: "El color es " + c1.obtenerColor()


        //*****   Creo un triángulo!!
        mostrar: "Ingresar color del triángulo:"
        ingresar: color
        mostrar: "Ingresar base del cuadrado:"
        ingresar: base
        mostrar: "Ingresar altura del cuadrado:"
        ingresar: altura

        privado Triangulo t1 nuevaInstancia Triangulo(color, base, altura)
        mostrar: "El área del triángulo " + t1.calculoArea("triangulo!!!")
        mostrar: "El color es " + t1.obtenerColor()
    fin metodo
fin metodo
```

Primera clase:

En este ejemplo, defino una clase abstracta (Figura) con un método abstracto (calculoArea()). Esta clase va a manejar lo referido al color de las figuras creadas, pero va a delegar en sus clases hijas el manejo del cálculo del área de cada figura concreta que se implemente.

Segunda y tercera clase:

Siguiendo con el ejemplo, vemos las figuras concretas que se pueden crear: el cuadrado y el triángulo. Ambos son tipos de figuras, por lo que la herencia tiene sentido. Ambas clases tiene un constructor sobrecargado que recibe diferentes parámetros: el color (atributo colorN), que va a servir para modificar el atributo "color" de la clase padre Figura. Además, otros parámetros necesario para poder hacer el cálculo.

Además, implementan el método "calculoArea": cada figura concreta va a saber cómo se calcula su propia área.

En el caso de la clase Cuadrado, lo que hace es multiplicar su lados ("lado * lado"). En el caso de la clase Triangulo, también calcula su propia área ("(base * altura) / 2").

Cuarta clase:

Luego, en la clase de integración, se crea un objeto del tipo Cuadrado, llamado "c1". Previamente se ingresan los datos necesario para poder crearlo: "color" y "lado". Ambos datos se envían como parámetro al constructor de la clase Cuadrado para se pueda crear correctamente el objeto.

Continuamos cuando se invocan a los métodos del objeto del "c1", para poder calcular el área y mostrar el color previamente ingresado.

Finalmente, se repite la operación para crear un triángulo.

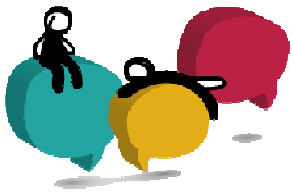
Centro de Elearning - FRBA - UTN

- Sobre clases, clases abstractas e interfaces

En relación a qué se pude y que no se puede hacer con las clases, clases abstractas, interfaces, atributos, métodos y sus modificadores de acceso.

	Clase	Abstracta	Interface
¿Puede tener atributos?	Si	Si	Si
¿Puede tener firmas de métodos con el modificador "abstracto"?	No	Si	No
¿Se puede instanciar / crear un objeto a partir de ella?	Si	No	No
¿Se puede declarar un atributo de ese tipo?	Si	Si	Si
¿Debe tener solo firmas de métodos?	No	No	Si
¿Puede tener métodos implementados (desarrollados)?	Si	Si	No
¿Debe llevar modificadores de acceso en atributos y métodos/firmas?	Si	Si	Si
¿Puede heredar de otra clase?	Si	Si	No
¿Puede heredar de otra interface?	No	No	Si
¿Puede heredar de una clase común e implementar una interface?	Si	Si	No
¿Puede heredar de una clase abstracta e implementar una interface?	Si	Si	No
¿Puede heredar de una clase abstracta?	Si	Si	No
¿Puede heredar de una clase no abstracta?	Si	Si	No
¿Puede implementar una interface?	Si	Si	No

FRBA - UTN



Actividad final de la Unidad 11

La consigna es desarrollar una pequeña aplicación sobre uno o ambos temas:

- Museo de Ciencias Naturales - Sección Botánica - Catálogo: Diseñar un sistema para la gestión del catálogo de plantas del museo, donde se puedan cargar datos de los mismos.
- Universidad Tecnológica Nacional - Facultad Regional Buenos Aires - Departamento de Empleados (pueden consultar la web de la UTN para mayor información): Diseñar un sistema para la gestión de empleados de la Universidad, donde se puedan manejar los datos personales y de sueldos.

Deberán incluir los siguientes conceptos:

- Interfaces
- Clases Abstractas
- Herencia Simple
- Modificadores de acceso
- Clase de test o integración (se debe llamar "**clase publica Test**" o indicar con un comentario cuál de todas es).

Aclaraciones:

Pueden usar 1 ejemplo de cada uno de estos conceptos.

Les dejo algunas ideas para resolver el ejercicio:

Por ejemplo, caso "otro museo".

Centro de Elearning - FRBA - U

1. Interfaz "GestionDeMuestras" con los métodos "crearMuestra" y "cambiarFechaMuestra".

2. Clase abstracta Museo, que permita: ponerle nombre o tipo al museo. Método abstracto "asignarDireccion" para la muestra.

3. Clases MuseoCiencias y MuseoRiver que hereden de Museo e implementan los métodos abstractos y que le ponga la dirección que corresponda a cada uno. Además, que implemente la interfaz y que permita cargar la fecha y tema a una muestra.

4. Clase "CreaMuestras" que instancie MuseoCiencias y MuseoRiver, permita crear muestras de cada uno y ponerles fecha. Pueden tener un menú con estas opciones en el constructor, por ejemplo. La idea de esta clase es ver cómo se integran todos los componentes anteriores, como "funciona" todo en conjunto, como se ensamblan las partes separadas en una lógica única y completa.

Esto es solamente un idea. El que desarrolle esto mismo tal cual o cambiándole únicamente los nombres NO será corregido. Pueden usar estas ideas como estructura base, pero siempre agregando (o sacando) algo.

Nuevamente: es obligatorio usar todos los conceptos mencionados en el enunciado Y la clase de test/integración mencionada en el punto 4. Sin esto, el trabajo no será corregido: será devuelto hasta que esté completo.

Tampoco se aceptarán entregas parciales.

RBA - UTN

Lo que vimos

- Conceptos de interfaces y clases abstractas.
- Conceptos de casting, o transformación de tipos de datos.
- El mecanismo del Polimorfismo.
- El concepto de clases internas.

Lo que viene:

- Los principales conceptos que hacen a la aplicación de los contenidos vistos en el curso en un ambiente laboral actual
- Los principales lenguajes de programación y plataformas más difundidos en la actualidad: PHP, C#, HTML, Javascript, Ruby y Ruby on Rails y Java
- Conceptos los patrones de diseño
- Concepto de ciclo de vida de un proyecto de desarrollo de software
- Ejercicio integrador resuelto

Centro de Elearning - FRBA - UTN