



UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

**Centro de
e-Learning**

FUNDAMENTOS DE LA PROGRAMACIÓN

Centro de E-learning - 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



MÓDULO 3 - UNIDAD 11

Conceptos Avanzados

Centro de e-Learning - FRBA - UTN



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.



Objetivos:

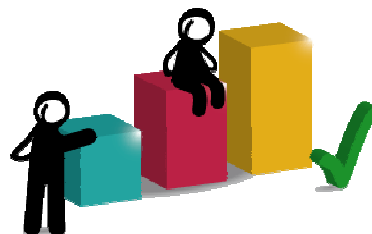
Que los participantes:

- 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.



Bloques temáticos:

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



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.



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.



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.



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



- 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.



ACTIVIDAD 1:

¿Qué otros ejemplos de Interfaces y sus implementaciones se les vienen a la mente?

Expresarlas en forma similar al ejemplo Animal, Hombre, Pez (no es necesario implementar los métodos en las clases, con dejar un comentario del tipo “// acá va la implementación” es suficiente.)

Con 1 interface y 2 o 3 clases es suficiente.

Fundamentar porqué el ejemplo dado requiere que una interface “imponga” el comportamiento al conjunto de clases.

¡RECUERDEN INDENTAR BIEN SU CÓDIGO!

Comparta sus respuestas en el foro de actividades de la Unidad.



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.



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 “sueldoConBeneficios”, ya que la forma de cálculo de los beneficios varía entre los tipos de empleados.

```
clase publica abstracta Empleado
    //atributos de la entidad
    privado Float sueldoBase
    privado Float sueldoConBeneficios
    //cálculos de negocio
    metodo publico Float sueldoBase()
        sueldoBase = 1000
        retornar: sueldoBase
    fin método
    metodo publico abstracto Float sueldoConBeneficios()
fin clase
```



```
clase publica Supervisor heredaDe Empleado
```

```
metodo publico Float sueldoConBeneficios()
```

```
    sueldoConBeneficios = sueldoBase * 1,2
```

```
    retornar: sueldoConBeneficios
```

```
fin metodo
```

```
fin clase
```

```
clase publica Gerente heredaDe Empleado
```

```
metodo publico Float sueldoConBeneficios()
```

```
    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.



ACTIVIDAD 2:

¿Qué otros ejemplos de clases abstractas y sus “herederas” se les vienen a la mente?

Expresarlas en forma similar al ejemplo Empleado, Supervisor, Gerente (no es necesario implementar los métodos en las clases, con dejar un comentario del tipo “// acá va la implementación” es suficiente).

Con 3 o 4 clases es suficiente.

¡RECUERDEN INDENTAR BIEN SU CÓDIGO!

Comparta sus respuestas en el foro de actividades de la Unidad.



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.



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

    metodo publico EjemploCast ()

        Integer numeroEntero

        Float numeroConComa

        numeroConComa = 3,14

        numeroEntero = (Integer)numeroConComa

        mostrar: “El número pasado a entero es “

        mostrar: numeroEntero

    fin metodo

fin clase
```



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

“El número pasado a entero es 3”

Ya que se transformó 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.



ACTIVIDAD 3:

Teniendo en cuenta los “tipos de datos” dados que venimos trabajando:

- String
- Integer
- Float
- Date
- Boolean

¿Cuáles tendría sentido castear de uno a hacia otro? Por ejemplo: de Integer a String. ¿Y cuáles no tendría sentido castear, dado que serían datos incompatibles? Por ejemplo: de String a Integer.

Expresar la forma y agregar algún ejemplo del estilo:

```
Integer i nuevaInstancia Integer()
```

```
i = 10
```

```
String s nuevaInstancia String()
```

```
s = (String) i
```

Comparta sus respuestas en el foro de actividades de la Unidad.



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 Integer  
    //esta interface va a tener el mismo comportamiento que la clase Integer  
fin interface
```

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



```
clase publica Suma heredaDe Operacion
    Integer resultado
    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
    Integer resultado

    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 ()
```

```
Operacion operacion1 nuevaInstancia Suma(1,2);
```

```
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 la clase Integer, 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:

Operacion operacion1 nuevaInstancia Suma(1,2);
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".

- 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.

Resultado resultado1 = operacion1. obtenerResultado ();

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

Integer resultado1 = operacion1. obtenerResultado ();

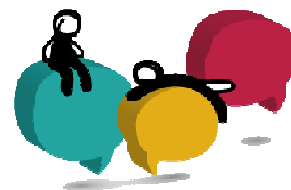
El polimorfismo, como se muestra en el ejemplo anterior, suele traer ventajas aplicado desde las interfaces, ya que permite crear nuevos tipos sin necesidad de tocar las clases ya existentes (imaginemos que deseamos añadir una clase "Restar", por ejemplo), basta con recompilar todo el código que incluye 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.



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
- Getters y Setters
- Métodos y atributos privados y públicos

Aclaraciones:

Pueden usar 1 ejemplo de cada uno de estos puntos.

No es necesario que implementen todos los métodos, el objetivo principal es que apliquen los temas nombrados con criterio.

Deberán subir la resolución en el foro de la unidad identificado como Actividad final.

Pueden optar por resolverlos o plantear mejoras/correcciones a los que vayan posteando.



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, ASP y Java
- Conceptos los patrones de diseño
- Concepto de ciclo de vida de un proyecto de desarrollo de software
- Ejercicio integrador

