UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

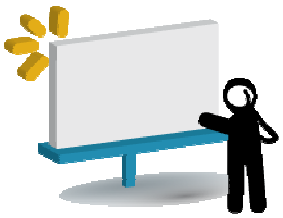
Centro de e-Learning

FUNDAMENTOS DE LA PROGRAMACIÓN

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



Presentación:

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

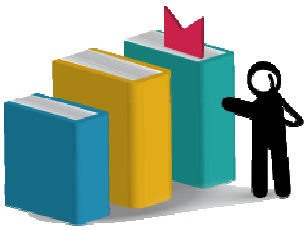
Así mismo se analiza otro de los mecanismos más poderosos del paradigma: el Polimorfismo, otro de los pilas de la POO.

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:

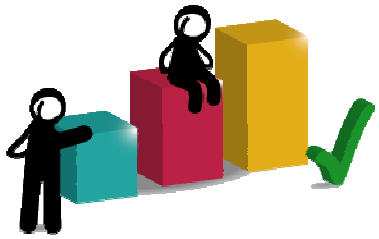
Que los participantes:

- 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:

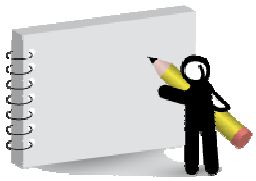
- Los foros proactivos asociados a cada una de las unidades.
- La Web 2.0.
- Los contextos de desempeño de los participantes.

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 DE ANÁLISIS 1:

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

Expresarlas en forma similar al ejemplo Animal, Hombre, Pez. Con una interface y dos o tres clases es suficiente para resolverlo.

Podrán comparar sus respuestas con los ejemplos resueltos en el "ANEXO 1 - Respuesta Actividad de Análisis 1" al final de esta unidad.

Previamente pueden consultar el "ANEXO 4 - Checklist de código".

IMPORTANTE:

- **¡Traten de resolverlo cada uno por sus propios medios!**
- **Las respuestas (el programa) NO debe ser enviado a los foros del Campus, salvo dudas puntuales de las respuestas.**

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 “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
```

```
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.

ACTIVIDAD DE ANÁLISIS 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.

Podrán comparar sus respuestas con los ejemplos resueltos en el "ANEXO 2 - Respuesta Actividad de Análisis 2" al final de esta unidad.

Previamente pueden consultar el "ANEXO 4 - Checklist de código".

IMPORTANTE:

- **¡Traten de resolverlo cada uno por sus propios medios!**
- **Las respuestas (el programa) NO debe ser enviado a los foros del Campus, salvo dudas puntuales de las respuestas.**

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
    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
```

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.

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 utilizando herencia:

```
clase publica abstracta Operacion
    protegido Integer resultado nuevaInstancia Integer()

    metodo publico abstracto realizarOperacion(Integer, Integer)

    metodo publico Integer obtenerResultado()
        retornar: resultado
    fin metodo
fin clase
```

```
clase publica Suma heredaDe Operacion
    metodo publico Suma()
    fin metodo

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

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

clase publica Multiplicacion heredaDe Operacion
    metodo publico Multiplicacion()
    fin metodo

    metodo publico Multiplicacion(Integer a, Integer b)
        realizarOperacion(a, b)
    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(10,20)
        Operacion operacion2 nuevaInstancia Multiplicacion(3,2)
        Suma operacion3 nuevaInstancia Suma()
        operacion3.realizarOperacion(100, 200)

        mostrar(operacion1) //muestra 30 --> (10+20)
        mostrar(operacion2) //muestra 6 --> (3*2)
        mostrar(operacion3) //muestra 300 --> (100+200)
    fin metodo

    metodo privado mostrar(Operacion op)
        mostrar: "El resultado es: " + op.obtenerResultado()
    fin metodo
fin clase
```

En este ejemplo tenemos:

- Clase Abstracta Operacion: define un método abstracto, que permitirá realizar cada operación concreta. Todas las operaciones concretas deberá heredar de esta clase.

- Clase Suma: hereda la clase abstracta Operación e implementa su método. La clase “Suma” es una “Operacion” que sólo sabe sumar dos números.

- Clase Multiplicacion: hereda la clase abstracta Operación e implementa su método. La clase “Multiplicacion” es un tipo de “Operacion” concreto que sólo sabe multiplicar dos números.

- Clase HagoOperacion: es una clase de test en la que podemos ver como se implementa el polimorfismo:

  Operacion operacion1 nuevaInstancia Suma(10,20)
  Operacion operacion2 nuevaInstancia Multiplicacion(3,2)

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

  Luego, ambos objetos se envían como parámetros a un método privado. Recordemos: "operacion1" ES una "Suma" pero, por la relación de herencia, también ES una "Operacion". Lo mismo ocurre con "operacion2".

  Esto nos permite tener un solo método privado "mostrar" cuyo parámetro es del tipo "Operacion":

  "metodo privado mostrar(**Operacion** op)"

  Sin polimorfismo, deberíamos tener métodos sobrecargados para cada tipo de objeto del parámetro:

  "metodo privado mostrar(**Suma** op)" y

  "metodo privado mostrar(**Multiplicacion** op)"

  Con "operacion3" ocurre algo similar: si bien es del tipo "Suma" y se instancia usando el constructor por defecto, ese objeto (al estar dentro de la

estructura de herencia) sigue siendo del tipo "Operacion", además de ser del tipo concreto "Suma", por lo que también podemos usarlo (a "operacion3" como parámetro para el "método mostrar(Operacion)".

Si necesitamos agregar una nueva operación, lo podríamos hacer así:

```
clase publica Division heredaDe Operacion
    metodo publico Division()
    fin metodo

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

    metodo publico realizarOperacion(Integer a, Integer b)
        si b != 0 entonces
            resultado = a / b
        sino
            mostrar: "No se puede dividir por 0"
            resultado = 0
        fin si
    fin metodo
fin clase
```

Haciendo algunos pequeños cambios en la clase de test para incorporar la nueva operación, tendríamos algo como esto:

```
clase publica HagoOperacion
    metodo publico HagoOperacion()
        Operacion operacion1 nuevaInstancia Suma(10,20)
        Operacion operacion2 nuevaInstancia Multiplicacion(3,2)
        Suma operacion3 nuevaInstancia Suma()
        operacion3.realizarOperacion(100, 200)
        Operacion operacion4 nuevaInstancia Division(10,2)  //se agrega

        mostrar(operacion1) //muestra 30 --> (10+20)
        mostrar(operacion2) //muestra 6 --> (3*2)
        mostrar(operacion3) //muestra 300 --> (100+200)
        mostrar(operacion4) //muestra 5 --> (10/2) // se agrega
    fin metodo

    metodo privado mostrar(Operacion op)
        mostrar: "El resultado es: " + op.obtenerResultado()
    fin metodo
fin clase
```

De esta forma solo agregamos un nuevo atributo y la invocación, pero el **método "mostrar" no cambia**. Eso se puede lograr solamente con polimorfismo.

Como se mencionó, el polimorfismo se puede implementar por herencia o por interfaces. En el ejemplo visto, se implementó utilizando herencia de clases. El mismo efecto se podría lograr utilizando interfaces, creando objetos del tipo de la interfaz, aunque instanciándo el atributo con un constructor (clase) concreto.

MiInterfaz atrib1 nuevaInstancia ClaseQueImplementaInterfaz()

MiInterfaz atrib2 nuevaInstancia OtraClaseQueImplementaInterfaz()

Tanto "atrib1" como "atrib2" son del tipo "MiInterfaz", aunque se instancian de diferente manera.

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 (de manera simplificada) 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.

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 "Integer" 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.

- 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.

- 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

ACTIVIDAD DE ANÁLISIS INTEGRADORA:

Requerimiento: 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.

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).

Por ejemplo, caso "otro museo":

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.

Pueden tomar esto como sugerencia o base para pensar el desarrollo.

Podrán comparar sus respuestas con los ejemplos resueltos en el "ANEXO 3 - Respuesta Actividad de Análisis Integradora" al final de esta unidad.

Previamente pueden consultar el "ANEXO 4 - Checklist de código".

IMPORTANTE:

- **¡Traten de resolverlo cada uno por sus propios medios!**
- **Las respuestas (el programa) NO debe ser enviado a los foros del Campus, salvo dudas puntuales de las respuestas.**

ANEXO 1 - Respuesta Actividad de Análisis 1

¿Trataron de resolver los requerimientos?

En caso afirmativo, avancen 3 páginas.

En caso negativo, ¡no avancen y traten de resolverlos antes de ver las respuestas!

Página dejada intencionalmente en blanco

Página dejada intencionalmente en blanco

Página dejada intencionalmente en blanco

Ejemplo 1)

```
interfaz publica Animal
    metodo publico patas()
fin interfaz

clase publico perro implementa Animal
    metodo publico patas()
        mostrar: "camina en 4 patas"
    fin metodo
fin clase

clase publico canguro implementa Animal
    metodo publico patas()
        mostrar: "camina en 2 patas"
    fin metodo
fin clase
```

Ejemplo 2)

```
interfaz publica Alarma
    metodo publico activa()
fin interfaz

clase publica Recordatorio implementa Alarma
    metodo publico activa()
        mostrar: "Recuerde presentar sus trabajos"
    fin metodo
fin clase

clase publica Despertador implementa Alarma
    metodo publico activa()
        mostrar: "Levantate!"
    fin metodo
fin clase

clase publica Llamada implementa Alarma
    metodo publico activa()
        mostrar: "Te estan llamando"
    fin metodo
fin clase
```

Ejemplo 3)

```
interfaz publica Monstruo
    metodo publico alimentacion(String)
fin interfaz

clase publica Vampiro implementa Monstruo
    metodo publico alimentacion(String cadaCuanto)
        mostrar: "Bebe sangre humana " + cadaCuanto
    fin metodo
fin clase

clase publica Zombie implementa Monstruo
    metodo publico alimentacion(String cadaCuanto)
        mostrar: "Desayuna, come, merienda y cena, cerebros " + cadaCuanto
    fin metodo
fin clase

clase publica HombreLobo implementa Monstruo
    metodo publico alimentacion(String cadaCuanto)
        mostrar: "Degusta carne humana en noches de Luna llena " +  cadaCuanto
    fin metodo
fin clase
```

Ejemplo 4)

```
interfaz publica HeroesMarvel
    metodo publico poderes()
fin interfaz

clase publica HombreArania implementa HeroesMarvel
    metodo publico poderes()
        mostrar: "Tiene las habilidades de una añara"
    fin metodo
fin clase

clase publica CapitanAmerica implementa HeroesMarvel
    metodo publico poderes()
        mostrar: "Humano mejorado, dicen... Aguante IRON MAN!!!!!"
    fin metodo
fin clase

clase publica Deadpool implementa HeroesMarvel
    metodo publico poderes()
        mostrar: "Puede autoregenerarse y tiene problemas con Wolverine"
    fin metodo
fin clase
```

Ejemplo 5)

```
interfaz publica Rock
    metodo publico guitarra()
fin interfaz

clase publica Heavy implementa Rock
    metodo publico guitarra()
        mostrar: "Guitarra distorsionada y potente. Solos con escalas a gran velocidad"
    fin metodo
fin clase

clase publica Reggae implementa Rock
    metodo publico guitarra()
        mostrar: "Guitarra con reverb y rasgueada. Solos con escala de blues, sincopados."
    fin metodo
fin clase

clase publica Sinfonico implementa Rock
    metodo publico guitarra ()
        mostrar: "Guitarras limpias y calidas. Solos precisos utilizando modos griegos"
    fin metodo
fin clase
```

Ejemplo 6)

```
interfaz publica Electricidad
    publico Integer consumo nuevaInstancia Integer()
    metodo publico Integer consumoElectrico()
fin interfaz

clase publica Microondas implementa Electricidad
    metodo publico Integer consumoElectrico()
        consumo = 500
        retonar: consumo
    fin metodo
fin clase

clase publica Computadora implementa Electricidad
    metodo publico Integer consumoElectrico()
        consumo = 650
        retonar: consumo
    fin metodo
fin clase

clase publica Televisor implementa Electricidad
    metodo publico Integer consumoElectrico()
        consumo = 250
        retonar: consumo
    fin metodo
fin clase
```

Ejemplo 7)

```
interfaz publica Libreria
    metodo publico utilesEscolares()
fin interfaz

clase publica Lapiz implementa Libreria
    metodo publico utilesEscolares()
        mostrar: "lapices de colores"
    fin metodo
fin clase

clase publica TipoRegla implementa Libreria
    metodo publico utilesEscolares()
        mostrar: "Regla graduada"
    fin metodo
fin clase
```

Ejemplo 8)

```
interfaz publica Puntuacion
    privado Integer puntos nuevaInstancia Integer()
    metodo publico anotarTanto()
fin interfaz

clase publica Futbolista implementa Puntuacion
    metodo publico anotarTanto()
        puntos = 1
    fin metodo
fin clase

clase publica Basquetbolista implementa Puntuacion
    metodo publico anotarTanto()
        String tipo nuevaInstancia String()
        ingresar: tipo
        si tipo = "doble" entonces
            puntos = 2
        sino si tipo = "tripe" entonces
            puntos = 3
        sino
            puntos = 1
        fin si
    fin metodo
fin clase
```

Ejemplo 9)

```
interfaz publica Universidades
    metodo privado universidad()
fin interfaz

clase publico UADE implementa Universidades
    metodo privado universidad()
        mostrar: "es una universidad privada"
    fin metodo
fin clase

clase publico UTN implementa Universidades
    metodo privado universidad()
        mostrar: "es una universidad pública"
    fin metodo
fin clase
```

Ejemplo 10)

```
interfaz publica Herramienta
    publico Integer peligrosidad nuevaInstancia Integer()
    metodo publico calificacion()
fin interfaz

clase publica Taladro implementa Herramienta
    metodo publico calificacion()
        peligrosidad = 5
    fin metodo
fin clase

clase publica Amoladora implementa Herramienta
    metodo publico calificacion()
        peligrosidad = 9
    fin metodo
fin clase

clase publica Martillo implementa Herramienta
    metodo publico calificacion()
        peligrosidad = 2
    fin metodo
fin clase
```

ANEXO 2 - Respuesta Actividad de Análisis 2

¿Trataron de resolver los requerimientos?

En caso afirmativo, avancen 3 páginas.

En caso negativo, ¡no avancen y traten de resolverlos antes de ver las respuestas!

Página dejada intencionalmente en blanco

Página dejada intencionalmente en blanco

Página dejada intencionalmente en blanco

Ejemplo 1)

```
clase publica abstracta Escuela
    privado String ingreso nuevainstancia String()
    protegido String salida nuevainstancia String()
    protegido Float cuota nuevainstancia Float()

    metodo publico String horarioDeingreso()
        ingreso = "8 Am"
        retornar: ingreso
    fin metodo

    metodo publico abstracto String horarioDeSalida()
    metodo publico abstracto Float cuotaEscuela()
fin clase

clase publica AlumnoJardin heredaDe Escuela
    metodo publico String horarioDeSalida()
        salida = "11 am"
        retonar: salida
    fin metodo
    metodo publico Float cuotaEscuela()
        cuota = 8000.45
        retornar: cuota
    fin metodo
fin clase

clase publica AlumnoPrimaria heredaDe Escuela
    metodo publico String horarioDeSalida()
        salida = "12 am"
        retornar: salida
    fin metodo
    metodo publico Float cuotaEscuela()
        cuota = 9500.67
        retornar: cuota
    fin metodo
fin clase

clase publica AlumnoSecundaria heredaDE Escuela
    metodo publico String horarioDeSalida()
        salida = "1 pm"
        retonar: salida
    fin metodo
    metodo publico Float cuotaEscuela()
        cuota = 10500.12
        retornar: cuota
    fin metodo
fin clase
```

Ejemplo 2)

```
clase publica abstracta Instrumento
    protegido String tipo nuevaInstancia String ()

    metodo publico String tipo()
        tipo = ""
        retornar: tipo
    fin metodo

    metodo publico abstracto String definirTipo()
    metodo publico abstracto Integer definirCantidadCuerdas()
fin clase

clase publica Guitarra heredaDe Instrumento
    metodo publico String definirTipo()
        tipo = "acústico"
        retornar: tipo
    fin metodo

    metodo publico Integer definirCantidadCuerdas()
        retornar: 6
    fin metodo
fin clase

clase publica Violin heredaDe Instrumento
    metodo publico String definirTipo ()
        tipo = "clásico"
        retornar: tipo
    fin metodo

    metodo publico Integer definirCantidadCuerdas()
        retornar: 4
    fin metodo
fin clase
```

Ejemplo 3)

```
clase publica abstracta Remera
    privado Float precio nuevaInstancia Float()
    protegido Float precioFinal nuevaInstancia Float()

    metodo publico Float precioBase()
        precio = 100.00
        retornar: precio
    fin metodo

    metodo publico abstracto Float calcularPrecioFinal()
fin clase

clase publica RemeraNacional heredaDe Remera
    metodo publico Float calcularPrecioFinal()
        precioFinal = precioBase() + 0.21 * precioBase()
        retornar: precioFinal
    fin metodo
fin clase

clase publica RemeraImportada heredaDe Remera
    metodo publico Float calcularPrecioFinal()
        precioFinal = precioBase() + 0.5 * precioBase()
        retornar: precioFinal
    fin metodo
fin clase

clase publica RemeraPersonalizada heredaDe Remera
    metodo publico Float calcularPrecioFinal()
        Float precioPersonalizacion nuevaInstancia Float()

        ingresar: precioPersonalizacion
        precioFinal = precioBase() + precioPersonalizacion
        retornar: precioFinal
    fin metodo
fin clase
```

Ejemplo 4)

```
clase publica abstracta modoDescongelar
    protegido Integer tiempoCero nuevaInstancia Integer()
    protegido Integer tiempoProgramado nuevaInstancia Integer()

    metodo protegido Integer resetearTiempo()
        tiempoCero = 0
        retornar: tiempoCero
    fin metodo

    metodo protegido abstracto Integer seteoDetiempo()
fin clase

clase publica Pescado heredaDe modoDescongelar
    metodo publico Integer seteoDetiempo()
        tiempoProgramado = tiempoCero + 10
        retornar: tiempoProgramado
    fin metodo
fin clase

clase publica Carne heredaDe modoDescongelar
    metodo publico Integer seteoDetiempo()
        tiempoProgramado = tiempoCero + 30
        retornar: tiempoProgramado
    fin metodo
fin clase

clase publica Pollo heredaDe modoDescongelar
    metodo publico Integer seteoDetiempo()
        tiempoProgramado = tiempoCero + 20
        retornar: tiempoProgramado
    fin metodo
fin clase
```

Ejemplo 5)

```
clase publica abstracta EntradaCine
    publico Float boletoComun nuevaInstancia Float()
    publico Float boletoConDescuento nuevaInstancia Float()

    metodo publico Float boletoComun()
        boletoComun = 6,50
        retornar: boletoComun
    fin metodo

    metodo publico abstracto Float calcularBoletoConDescuento()
fin clase

clase publica BoletoNiño heredaDe EntradaCine
    metodo publico Float calcularBoletoConDescuento()
        boletoConDescuento = boletoComun * 1,5
        retornar: boletoConDescuento
    fin metodo
fin clase

clase publica BoletoTerceraEdad heredaDe EntradaCine
    metodo publico Float calcularBoletoConDescuento()
        boletoConDescuento = boletoComun * 2,5
        retornar: boletoConDescuento
    fin metodo
fin clase
```

Ejemplo 6)

```
clase publica abstracta Bombero
    protegido Float sinSueldo nuevaInstancia Float ()

    metodo publico Float sinSueldo()
        sinSueldo = 0.00
        retornar: sinSueldo
    fin metodo

    metodo publico abstracto Float calcularSueldo ()
fin clase

clase publica Jubilado heredaDe Bombero
    metodo publico Float calcularSueldo ()
        Float sueldoJubilado nuevaInstancia Float ()
        sueldoJubilado = 40000.00
        retornar: sueldoJubilado
    fin metodo
fin clase

clase publica Voluntario heredaDe Bombero
    metodo publico Float calcularSueldo ()
        sinSueldo = 0.00
        retornar: sinSueldo
    fin metodo
fin clase
```

Ejemplo 7)

```
clase publica abstacta AutoMovil
    privado String marca nuevaInstancia String()
    protegido Float precio nuevaInstancia Float()

    metodo publico String marca()
        marca = "Nikola"
        retornar: marca
    fin metodo

    metodo publico abstracto Float calculaPrecioDeAutomovil()
fin clase

clase publica Camioneta4x4 heredaDe AutoMovil
    metodo publico Float calculaPrecioDeAutomovil()
        precio = 100.00
        retornar: precio
    fin metodo
fin clase

clase publica AutoDiesel heredaDe AutoMovil
    metodo publico Float calculaPrecioDeAutomovil()
        precio = 80.00
        retornar: precio
    fin metodo
fin clase

clase publica AutoElectrico heredaDe AutoMovil
    metodo publico Float calculaPrecioDeAutomovil()
        precio = 120.00
        retornar: precio
    fin metodo
fin clase
```

Ejemplo 8)

```
clase publica abstracta Felino
    protegido String tamanio nuevaInstancia String ()
    protegido Boolean esDomestico nuevaInstancia Boolean()

    metodo publico String tamanio()
        retornar: tamanio
    fin metodo

    metodo publico Boolean obtenerEsDomestico()
        retornar: esDomestico
    fin metodo

    metodo publico abstracto datosFelino()
fin clase

clase publica Gato heredaDe Felino
    metodo publico datosFelino()
        tamanio = "chico"
        esDomestico = verdadero
    fin metodo
fin clase

clase publica Tigre heredaDe Felino
    metodo publico datosFelino()
        tamanio = "mediano"
        esDomestico = falso
    fin metodo
fin clase

clase publica Leon heredaDe Felino
    metodo publico datosFelino()
        tamanio = "grande"
        esDomestico = falso
    fin metodo
fin clase

clase publica Pantera heredaDe Felino
    metodo publico datosFelino()
        tamanio = "mediano a grande"
        esDomestico = falso
    fin metodo
fin clase
```

Ejemplo 9)

```
clase publica abstracta Leche
    protegido Float grasasTrans nuevaInstacia Float()
    protegido Float grasasSaturadas nuevaInstancia Float()
    protegido Float grasasTotales nuevaInstancia Float()

    metodo publico abstracto Float calculoDeGrasasTotales()
fin clase

clase publica SerenisimaDescremada heredaDe Leche
    metodo publico Float calculoDeGrasasTotales()
        grasasTotales = grasasTrans + grasasSaturadas * 1.2
        retornar: grasasTotales
    fin metodo

fin clase

clase publica SancorDescremada heredaDe Leche
    metodo publico Float calculoDeGrasasTotales()
        grasasTotales = grasasTrans + grasasSaturadas * 0.9
        retornar: grasasTotales
    fin metodo
fin clase

clase publica IlolayDescremada heredaDe Leche
    metodo publico Float calculoDeGrasasTotales()
        grasasTotales = grasasTrans + grasasSaturadas * 1.3
        retornar: grasasTotales
    fin metodo
fin clase
```

Ejemplo 10)

```
clase publica abstracta AutoMovil
    publico Integer litrosEnElTanque nuevaInstancia Integer()

    metodo publico arrancar()
        mostrar: "arrancó"
    fin metodo

    metodo publico abstracto llenarTanque()
fin clase

clase publica Camioneta heredaDe Automovil
    metodo publico llenarTanque()
        litrosEnElTanque = litrosEnElTanque + 100
        mostrar: "serían 3600 pesos"
    fin metodo
fin clase

clase publica AutoPequenio heredaDe Automovil
    metodo publico llenarTanque()
        litrosEnElTanque = litrosEnElTanque + 40
        mostrar: "serian 1500 pesos"
    fin metodo
fin clase
```

ANEXO 3 - Respuesta Actividad de Análisis Integradora

¿Trataron de resolver los requerimientos?

En caso afirmativo, avancen 3 páginas.

En caso negativo, ¡no avancen y traten de resolverlos antes de ver las respuestas!

Página dejada intencionalmente en blanco

Página dejada intencionalmente en blanco

Página dejada intencionalmente en blanco

Ejemplo 1)

```
interfaz publica AreasDeCultura
    metodo publico contenidoDeLaMuestra(Catalogo)
    metodo publico mostrarTipoOrganizacion()
fin interfaz

clase publica Museo implementa AreasDeCultura
    privado Catalogo cat nuevaInstancia CatalogoDePlantas()

    metodo publico contenidoDeLaMuestra(Catalogo c)
        cat = c
    fin metodo

    metodo publico mostrarTipoOrganizacion()
        mostrar: "Esta organización es un Museo"
    fin metodo
fin clase

clase publica Planta
    privado String nom nuevaInstancia String()
    privado Boolean esAromatica nuevaInstancia Boolean()

    metodo publico asignarNombre(String nombrePlanta)
        nom = nombrePlanta
    fin metodo

    metodo publico asignaresAromatica(Boolean plantaAromatica)
        esAromatica = plantaAromatica
    fin metodo
fin clase

clase publica abstracta Catalogo
    protegido Plantas catalogo[10000] nuevaInstancia Plantas()
    protegido Integer i nuevaInstancia Integer()

    metodo publico llenarCatalogo(Planta p)
        catalogo[i] = p
        i = i + 1
    fin metodo

    metodo publico abstracto mostrarTipoDeCatalogo()
fin clase
```

```
clase publica CatalogoDePlantas heredaDe Catalogo
    metodo publico CatalogoDePlantas()
        i = 1
    fin metodo

    metodo publico mostrarTipoDeCatalogo()
        mostrar: "Este es el catálogo principal para plantas"
    fin metodo
fin clase

clase publica Test
    metodo publico Test()
        Planta planta1 nuevaInstancia Planta()
        Planta planta2 nuevaInstancia Planta()

        planta1.asignarNombre("jazmin")
        planta1.asignaresAromatica(verdadero)
        planta2.asignarNombre("bignonia")
        planta2.asignaresAromatica(falso)

        CatalogoDePlantas c nuevaInstancia CatalogoDePlantas()
        c.mostrarTipoDeCatalogo()
        c.llenarCatalogo(planta1)
        c.llenarCatalogo(planta2)

        Museo m nuevaInstancia Museo()
        m.contenidoDeLaMuestra(c)
    fin metodo
fin clase
```

Ejemplo 2)

```
interfaz publica MuseoDeBotanica
    metodo publico ingresaAreaDeBotanica (String)
    metodo publico diasDeVisitasAlAreaDeBotanica ()
    metodo publico Integer preciosDeEntradas ()
fin interfaz

clase publica abstracta AreadeBotanica implementa MuseoDeBotanica

    protegido String area nuevaInstancia String ()

    metodo publico AreaDeBotanica ()
        area = ""
    fin metodo

    metodo publico ingresaAreaDeBotanica (String areaNueva)
        area = areaNueva
    fin metodo

    metodo publico String retornaAreaDeBotanica ()
        retorna: area
    fin metodo

    metodo publico diasDevisitasAlAreaDeBotanica ()
        String dia nuevaInstancia String ()

        mostrar: "Ingrese día"
        ingresar: dia

        en caso de dia hacer
            caso = "Lunes"
                mostrar: "Horario de visitas de 10 a 13 horas"
            fin caso
            caso = "Miércoles"
                mostrar: "Horario de visitas de 16 a 18 horas"
            fin caso
            caso = "Viernes"
                    mostrar: "Horario de visitas a las 19 horas"
            fin caso
            caso = "Sábado"
                mostrar: "Horario de 10 a 19:30 horas"
            fin caso
            sino
                mostrar: "No están programadas visitas al área de botánica el día ingresado"
        fin en caso de
    fin metodo
```

```
    metodo publico Integer preciosDeEntradas ()
        Integer valor nuevaInstancia Integer ()
        Integer edad nuevaInstancia Integer ()

        mostrar: "Ingrese su edad"
        ingresar: edad

        si edad > = 13 entonces
            valor = 250
        sino
            valor = 150
        fin si
        retornar: valor
    fin metodo

    metodo publico abstracto areaQueVisita ()
fin clase

clase publica DivisionPlantasCelulares heredaDe AreaDeBotanica ()

    metodo publico areaQueVisita ()
        si area = "Plantas Celulares" entonces
            mostrar: "Las muestras son los días Lunes y Miércoles"
        fin si
    fin metodo
fin clase

clase publica DivisionPlantasVasculares heredaDe AreaDeBotanica ()

    metodo publico areaQueVisita ()
        si area = "Plantas Vasculares" entonces
            mostrar: "Las muestras son los días Viernes y Sábados"
        fin si
    fin metodo
fin clase
```

```
clase publica TestMuseo ()
    metodo publico TestMuseo ()

        privado AreadeBotanica visita1 nuevaInstancia DivisionPlantasCelulares ()
        privado AreaDeBotanica visita2 nuevaInstancia DivisionPlantasVasculares ()

        visita1.ingresaAreaDeBotanica ("Plantas Celulares")
        mostrar: visita1.retornaAreaDeBotanica ()
        visita1.diasDeVisitasAlAreaDeBotanica ()
        mostrar: visita1.preciosDeEntradas ()

        visita2.ingresaAreaDeBotanica ("Plantas Vasculares")
        mostrar: visita2.retornaAreaDeBotanica ()
        visita2.diasDeVisitasAlAreaDeBotanica ()
        mostrar: visita2.preciosDeEntradas ()

    fin metodo
fin clase
```

Ejemplo 3)

```
interfaz publica Datos
    metodo publico ingresarNombre(String)
    metodo publico ingresarUnidades(Integer)
    metodo publico ingresarOrigen(String)
    metodo publico ingresarIncorporacion(String)
fin interfaz

clase privada abstracta Planta implementa Datos
    protegido String nombreTecnico nuevaInstancia String()
    protegido String origen nuevaInstancia String()
    protegido Integer nroEjemplaresAlmacen nuevaInstancia Integer()
    protegido String division nuevaInstancia String()
    protegido String tenerFlores nuevaInstancia String()
    protegido String tenerFrutos nuevaInstancia String()
    protegido Date fecha nuevaInstancia Date()

    metodo publico ingresarNombre(String nombreRecibido)
        nombreTecnico = nombreRecibido
    fin metodo
    metodo publico ingresarUnidades(Integer cantidadRecibida)
        nroEjemplaresAlmacen = cantidadRecibida
    fin metodo
    metodo publico ingresarOrigen(String origenRecibido)
        origen = origenRecibido
    fin metodo
    metodo publico ingresarIncorporacion(String incorporacion)
        fecha = (Date)incorporacion
    fin metodo

    metodo publico abstracto ingresarPigmentacion()
    metodo publico abstracto mostrarDatos()

    metodo protegido mostrar()
        mostrar: "Nombre de la planta: " + nombreTecnico
        mostrar: "Lugar de origen: " + origen
        mostrar: "Unidades Registradas en la instalacion: " + (String)nroEjemplares
        mostrar: "-Informacion de la especie-"
    fin metodo

    metodo protegido Planta()
        nombreTecnico = ""
        origen = ""
        nroEjemplaresAlmacen = 0
        division = ""
    fin metodo
fin clase
```

```
clase publica Helecho heredaDe Planta
    metodo publico Helecho()
        tenerFlores = "No posee flores"
        tenerFrutos = "No posee frutos"
        division = "Planta-Vascular"
    fin metodo

    metodo publico ingresarPigmentacion()
        mostrar = "No se puede ingresar, los helechos son en su mayoria todos son de coloracion verde"
    fin metodo

    metodo publico mostrarDatos()
        mostrar()
        mostrar: "Los helechos tienen hojas de color verde, pertenece a la division: " + division + ", " + tenerFlores + " y " + tenerFrutos
    fin metodo
fin clase

clase publica Angiosperma heredaDe Planta
    publico String colorFlor nuevaInstancia String()
    publico String nombreFlor nuevaInstancia String()

    metodo publico Angiosperma()
        tenerFlores = "Si posee Flores"
        tenerFrutos = "Si posee Frutos"
        division = "Planta-Vascular"
    fin metodo

    metodo publico ingresarPigmentacion()
        mostrar: "Ingrese el color que tienen las flores"
        ingresar: colorFlor
    fin metodo

    metodo publico ingresarNombreFlor()
        mostrar: "Ingrese el nombre que tienen las flores"
        ingresar: nombreFlor
    fin metodo

    metodo publico mostrarDatos()
        mostrar()
        mostrar: "Pertenece a la division" + division + ", " + tenerFlores + " y " + tenerFrutos + ". El nombre de la flor: " +
            nombreFlor + ". El color caracteristico de la flor es: " + colorFlor
    fin metodo
fin clase
```

```
clase publica ClasePublicaTest
    metodo publico ClasePublicaTest()
        publico Helecho ejemploHelechoSanluis nuevaInstancia Helecho()
        publico Angiosperma ejemploLimonero nuevaInstancia Angiosperma()

        ejemploHelechoSanluis.ingresarNombre("Pteridium arachnoideum")
        ejemploHelechoSanluis.ingresarUnidades(20)
        ejemploHelechoSanluis.ingresarIncorporacion("16/12/2020")
        ejemploHelechoSanluis.ingresarOrigen("San Luis")

        ejemploLimonero.ingresarNombre("CitrusÃ–Limon")
        ejemploLimonero.ingresarUnidades(10)
        ejemploLimonero.ingresarIncorporacion("16/12/2020")
        ejemploLimonero.ingresarOrigen("Tucuman")
        ejemploLimonero.ingresarNombreFlor()
        ejemploLimonero.ingresarPigmentacion()

        ejemploHelechoSanluis.mostrarDatos()
        ejemploLimonero.mostrarDatos()
    fin metodo
fin clase
```

Ejemplo 4)

```
clase publico abstracta MuseoDeCienciasNaturales
    publico String nombre nuevaInstancia String ()
    publico String tipo nuevaInstancia String()

    metodo publico MuseoDeCienciasNaturales()
        nombre = "Rito"
        tipo = "Ciencias Naturales - Botanica"
    fin metodo

    metodo publico cargarNombre (String nNombre)
        nombre = nNombre
    fin metodo

    metodo publico String devolverNombre ()
        retornar: nombre
    fin metodo

    metodo publico cargarTipo (String nTipo)
        tipo = nTipo
    fin metodo

    metodo publico String devolverTipo ()
        retornar: tipo
    fin metodo

    metodo publico abstracto asignarSala (Integer)
    metodo publico abstracto AsignarTipo (String)
fin clase

clase publica CatalogoBotanica heredaDe MuseoDeCienciasNaturales, implementa Planta

    privado Integer sala nuevaInstancia Integer ()
    privado String nombrePlanta nuevaInstancia String ()
    privado String tipoPlanta nuevaInstancia String ()
    privado String tipoSala nuevaInstancia String ()

    metodo publico asignarSala (Integer nSala)
        sala = nSala
    fin metodo
```

```
    metodo publico Integer devolverSala ()
        retornar : sala
    fin metodo

    metodo publico asignarTipoSala (String nuevoTipoSala)
        tipoSala = nuevoTipoSala
    fin metodo

    metodo publico String devolverTipoSala ()
        retornar: tipoSala
    fin metodo

    metodo publico crearPlanta (String nPlanta)
        nombrePlanta = nPlanta
    fin metodo

    metodo publico String devolverNombrePlanta ()
        retornar: nombrePlanta
    fin metodo

    metodo publico asignarTipoPlanta (String nuevoTipoPlanta)
        tipoPlanta = nuevoTipoPlanta
    fin metodo

    metodo publico String devolverTipoPlanta ()
        retornar: tipoPlanta
    fin metodo

fin clase

interfaz publico Planta
    metodo publico crearPlanta (String)
    metodo publico asignarTipoPlanta (String)
fin interfaz
```

```
clase publica Test
    metodo publico Test ()
        privado CatalogoBotanica planta1 nuevaInstancia CatalogoBotanica ()
        privado CatalogoBotanica planta2 nuevaInstancia CatalogoBotanica ()

        planta1.crearPlanta ("Suculenta")
        planta1.asignarTipoPlanta ("Terrestre")
        planta1.asignarSala (4)
        planta1.asignarTipoSala ("Botanica")

        planta2.crearPlanta ("Rosa")
        planta2.asignarTipoPlanta ("Terrestre")
        planta2.asignarSala (4)
        planta2.asignarTipoSala ("Botanica Terrestre")
    fin metodo
fin clase
```

Ejemplo 5)

```
interfaz publica CaracteristicasMuseo
    metodo publico ubicacion()
    metodo publico horariosDeApertura(Integer, Integer)
fin interfaz

clase publica abstracta MaterialDeColeccionBotanica

    protegido String seccion nuevaInstancia String()
    protegido String materialMuestra nuevaInstancia String()

    metodo publico seccionDelMaterial()
        seccion = "Botánica"
    fin metodo

    metodo publico abstracto asignarFechaDeColecta(Date)

    metodo publico tipoDeMuestra(String parteDePlanta)
        materialMuestra = parteDePlanta
    fin metodo

fin clase

clase publica MuestraBotanica heredaDe MaterialDeColeccionBotanica

    privado Date fechaDeColecta nuevaInstancia Date()
    privado String especie nuevaInstancia String()

    metodo publico MuestraBotanica(String nombreCientifico)
        especie = nombreCientifico
    fin metodo

    metodo publico asignarFechaDeColecta(Date fecha)
        fechaDeColecta = fecha
    fin metodo

    metodo publico verCaracteristicasMuestra()
        mostrar: "Sección de la muestra: " + seccion
        mostrar: "Especie: " + especie
        mostrar: "Fecha de colecta: " + fechaDeColecta
        mostrar: "Corresponde a la parte de la planta: " + materialMuestra
    fin metodo
fin clase
```

```
clase publica MuseoDeCiencias implementa CaracteristicasMuseo
    privado String nombreMuseo nuevaInstancia String()
    privado String categoriaMuseo nuevaInstancia String()
    privado Integer horarioApertura nuevaInstancia Integer()
    privado Integer horarioCierre nuevaInstancia Integer()
    privado String ciudadMuseo nuevaInstancia String()
    privado String paisMuseo nuevaInstancia String()

    metodo publico MuseoDeCiencias(String nombre)
        nombreMuseo = nombre
        categoriaMuseo = "Ciencias Naturales"
        mostrar: "Bienvenidos al museo" + nombreMuseo
    fin metodo

    metodo publico verCategoriaDelMuseo()
        mostrar: categoriaMuseo
    fin metodo

    metodo publico ubicacion()
        mostrar: "Por favor, ingrese el país donde se ubica el museo: "
        ingresar: paisMuseo

        mostrar: "Por favor, ingrese la ciudad donde se ubica el museo: "
        ingresar: ciudadMuseo
    fin metodo

    metodo publico horariosDeApertura(Integer apertura, Integer cierre)
        si apertura >= 0 Y apertura <= 23 entonces
            horarioApertura = apertura
        sino
            mostrar: "Error. Deben ser números dentro del rango horario."
        fin si
        en caso de cierre hacer
            caso < 24 Y caso >= 0
                horarioCierre = cierre
            fin caso
            caso > 24
                mostrar: "El horario de cierre debe encontrarse entre 0 y 23"
            fin caso
            caso < 0
                mostrar: "El horario de cierre no puede ser un número negativo."
            fin caso
        fin en caso
    fin metodo
```

```
    metodo publico verHorarioApertura()
        mostrar: "El Museo " + nombreMuseo + "abre a las: "
        mostrar: horarioApertura
    fin metodo

    metodo publico verHorarioCierre()
        mostrar: "El Museo " + nombreMuseo + "abre a las: "
        mostrar: horarioCierre
    fin metodo

    metodo publico verUbicacion()
        mostrar: "El museo " + nombreMuseo + "se encuentra ubicado en " + paisMuseo + "en la ciudad de " + ciudadMuseo + "."
    fin metodo

fin clase

clase publica TestDeMuseo
    metodo publico TestDeMuseo
        MuseoDeCiencias nuevoMuseo nuevaInstancia MuseoDeCiencias("Bernardino Rivadavia")
        nuevoMuseo.ubicacion()
        nuevoMuseo.horariosDeApertura(10, 19)
        nuevoMuseo.verHorarioApertura()
        nuevoMuseo.verHorarioCierre()
        nuevoMuseo.verUbicacion()
        nuevoMuseo.verCategoriaDelMuseo()

        MuestraBotanica troncoFosil nuevaInstancia MuestraBotanica("Araucaria mirabilis")
        troncoFosil.seccionDelMaterial()
        troncoFosil.tipoDeMuestra("Tallo")
        troncoFosil.asignarFechaDeColecta(12/06/1995)
        troncoFosil.verCaracteristicasMuestra()
    fin metodo
fin clase
```

Ejemplo 6)

```
clase publica Plantas heredaDe Botanica
    privado String familia nuevaInstancia String()
    privado String genero nuevaInstancia String()
    privado String especie nuevaInstancia String()
    privado String nombrePlanta nuevaInstancia String()

    metodo publico seccionBotanica()
        mostrar: "las plantas se dividen por categorias (nombre,familia, genero y especie)"
    fin metodo

    metodo publico definirNombrePlanta(String nuevoNombre)
        nombrePlanta = nuevoNombre
        mostrar: "el nombre de la planta es: " + nombrePlanta
    fin metodo
    metodo publico definirFamilia(String nuevaFamilia)
        familia = nuevaFamilia
    fin metodo
    metodo publico definirGenero(String nuevoGenero)
        genero = nuevoGenero
    fin metodo
    metodo publico definirEspecie(String nuevaEspecie)
        especie = nuevaEspecie
    fin metodo
    metodo publico String obtenerFamilia()
        retornar: familia
    fin metodo
    metodo publico String obtenerGenero()
        retornar: genero
    fin metodo
    metodo publico String obtenerEspecie()
        retornar: especie
    fin metodo
    metodo publico String obtenerNombrePlanta()
        retornar: nombrePlanta
    fin metodo
fin clase
```

```
interfaz publica Museo
    metodo publico mostrarSeccion()
    metodo publico String obtenerNombreMuseo()
fin interfaz

clase publica abstracta Botanica implementa Museo
    metodo publico mostrarSeccion()
        mostrar: "sección de botanica"
    fin metodo
    metodo publico String obtenerNombreMuseo()
        retornar: "Museo Argentino de Ciencias Naturales"
    fin metodo
    metodo publico abstracto seccionBotanica()
fin clase

clase publica Test //clase test
    metodo publico Test()
        Plantas planta1 nuevaInstancia Plantas()
        Plantas planta2 nuevaInstancia Plantas()

        planta1.seccionBotanica()
        planta1.definirNombrePlanta("ROSA")
        planta1.definirFamilia("ROSACEAE")
        planta1.definirGenero("ROSA")
        planta1.definirEspecie("ROSA SPP")

        planta2.seccionBotanica()
        planta2.definirNombrePlanta("CLAVEL")
        planta2.definirFamilia("CARYOPHYLLIDAE")
        planta2.definirGenero("DIANTHUS")
        planta2.definirEspecie("DIANTHUS CARYOPHYLLUS")
    fin metodo
fin clase
```

Ejemplo 7)

```
interfaz publica DefinicionPlanta
    publico String definicion nuevaInstancia String()
    metodo publico definicionP()
fin interfaz

clase publica abstracta Planta implementa DefinicionPlanta
    protegido String nombrePlanta nuevaInstancia String()
    protegido String tipoPlanta nuevaInstancia String()

    metodo publico Planta()
        mostrar: "Catálogo de Plantas"
    fin metodo

    metodo publico definicionP()
        definicion = "ser vivo fotosintético sin capacidad locomotora"
    fin metodo

    metodo publico abstracto tipoPlanta (String)
fin clase

clase publica SegunTamanio heredaDe Planta
    privado String tamanio nuevaInstancia String()

    metodo publico SegunTamanio()
    fin metodo

    metodo publico tamanio (String nuevoTamanio)
        tamanio = nuevoTamanio
    fin metodo

    metodo publico tipoPlanta (String nuevoTipo)
        tipoPlanta = nuevoTipo
    fin metodo
fin clase

clase publica Test
    metodo publico Test()
        SegunTamanio planta1 nuevaInstancia SegunTamanio()
        planta1.tamanio("muy grande")
        planta1.tipoPlanta("arbol")

        SegunTamanio planta2 nuevaInstancia SegunTamanio()
        planta2.tamanio("pequeña")
        planta2.tipoPlanta("mata")
    fin metodo
fin clase
```

ANEXO 4 - Checklist de código

Algunas consideraciones a la hora de revisar su propio código que se suman al checklist de la unidad anterior:

	Cumple	
Tema	**Si**	**No**
Las interfaces tiene únicamente atributos y firmas de métodos y no tienen métodos implementados (desarrollados)		
Una clase que implementa una interfaz, está implementando todos los métodos cuyas firmas aparecen en la interfaz, SIN excepciones, TODOS los métodos		
Las clases abstractas tiene el modificador "abstracta" en la declaración de la clase		
Las clases abstractas tienen, al menos, un método abstracto		
Los métodos abstractos no están implementados, sino que solamente tienen la firma del método (incluidos parámetros y tipo que retorna) y el modificador "abstracto".		
La clase que hereda de una clase abstracta implementa TODOS los métodos abstractos de la clase abstracta		
La firma de los métodos abstractos (en la clase abstracta) es la misma firma que la de los métodos implementados en la clase hija.		
Se verifica que los tipos de datos en el casting no impliquen tipos incompatibles (String a Integer, por ejemplo) ni pérdida de precisión (Float a Integer).		
Se verifica que no hay métodos abstractos en clases NO abstractas		
Los métodos abstractos implementados (en la clase hija) no llevan el modificador "abstracto".		

Si todas las respuestas son afirmativas, hay muy altas chances que el programa esté libre de errores de código.

ANEXO 5 - Preguntas frecuentes

1) Pregunta: ¿Qué relación hay entre polimorfismo y herencia? ¿Tienen sentido entre sí? ¿Es posible mezclar estos dos conceptos?

Respuesta: Si, es posible. Hacer esto sería perfectamente válido:

```
clase publica Persona
   publico String tipo nuevaInstancia String()
   metodo publico Persona()
   fin metodo
fin clase

clase publica Humano heredaDe Persona
   metodo publico Humano()
      tipo = "persona humana, por ejemplo, un alumno"
   fin metodo
fin clase

clase publica NoHumano heredaDe Persona
   metodo publico NoHumano()
      tipo = "persona NO humana, por ejemplo, un chimpancé"
   fin metodo
fin clase
```

```
clase publica Test
  metodo publico Test()
    Humano h1 nuevaInstancia Humano()
    NoHumano nh nuevaInstancia NoHumano()
    Persona p1 nuevaInstancia Humano()
    Persona p2 nuevaInstancia NoHumano()

    muestroPersona(h1)
    muestroPersona(nh)
    muestroPersona(p1)
    muestroPersona(p2)
  fin metodo

  metodo privado muestroPersona(Persona p)
    mostrar: p.tipo
  fin metodo
fin clase
```

Explicación:

- las clase "Humano" y "NoHumano" heredan de "Persona".

- por lo tanto, al instanciar esas clases, los objetos que se crean son del tipo específico ("Humano" y "NoHumano"). Y, además, también son del tipo no específico "Persona" por la relación de herencia.

- dicho esto, en el "*metodo privado muestroPersona(Persona p)*" el objeto "p" es polimórfico porque ES del tipo Persona, pero además PUEDE SER tanto del tipo "Humano" como "NoHumano".

Adicionalmente se presentan algunos ejemplos de polimorfismo desde el punto de vista de las interfaces. PERO sin interfaces, con herencia, se puede lograr algo parecido.

2) P: "¿No es posible instanciar las interfaces, pero si se pueden declarar atributos de ese tipo?"

R: Correcto, porque la creación de la instancia es la declaración del atributo donde no solo de declara el tipo sino el CONSTRUCTOR utilizado, que es quién finalmente realiza la instanciación.

3) P: "¿Las interfaces declaran el método constructor? "

R: No, la interfaz no tiene constructor, porque no se puede instanciar. Además, cuando se programa la interface es probable que las clases aún no existan, por lo tanto no sabríamos qué constructores tienen. Adicionalmente, ¿si declaramos constructores de las clases en una interfaz, todas las clases que implementen la interfaz deberían implementar todos los constructores? No sería viable...

4) P: ¿ Cuál sería un ejemplo práctico de cuándo conviene hacer interfaces o Clases abstractas?

R: Se pueden usar interfaces cuando se quiere que, por ejemplo, un grupo de entidades cumplan sí o sí con cierto comportamiento mínimo en forma obligatoria: las personas tiene nombre, sean personas físicas o no.

También podemos pensar que a todas las formas geométricas se les puede calcular el área, pero cada una lo calcula de diferente forma.

Por otro lado, todos los edificios son construcciones, pero existen diferentes tipos como casas, apartamentos, dúplex, etc. y todas tienen, por ejemplo, una identificación (dirección), ubicación, tamaño...

Con las clases abstractas se pueden pensar algo de forma similar, teniendo presente que a los tipos concretos (clases hijas) ya se le puede agregar alguna funcionalidad resuelta (métodos implementados) y pedirles que las clase hijas resuelvan su funcionamiento de la forma que más les sirve a cada una (métodos abstractos).

5) P: Sobre la utilidad de una interface: ¿es un "recordatorio" para que otras clases la implementen? Y ¿cuál sería su utilidad práctica? ¿Es evitable su uso?

R: Supongamos que necesitamos que cortar una pieza de cartón de esta forma:

Con tomar el cartón, tijeras y haciendo los cortes sería suficiente, ¿es verdad? Puede quedar bien o no tan bien, pero no parece tan difícil.

Ahora bien, supongamos ahora no es una, sino que es necesario cortar 20 de esas formas y solo tenemos los cartones, una tijera y buscamos una hoja de papel: ¿no sería más sensato hacer un molde con el papel y luego cortar los 20 cartones siguiendo el patrón? ¿O seguiríamos haciendo los 20 cortes a ojo?

Siendo un poco más específicos, si tenemos el requerimiento para programar una máquina de estados (https://es.wikipedia.org/wiki/M%C3%A1quina_de_estados) con 10 estados, donde todos los estados tiene "cambio de estado", "volver al estado anterior", entre otros comportamientos y datos en común, ¿estamos seguro que vamos de que nos vamos a acordar de colocar TODOS los métodos y atributos en las 10 clases? ¿Y si fueran 30 estados? O

Contar con un mecanismo que evite dejar funcionalidades a la buena voluntad y memoria de los programadores es, al menos, riesgoso. Y todo riesgo debe mitigarse, siempre.

6) P: ¿Una clase puede heredar de una clase padre y, a su vez, esa misma clase (la hija) puede implementar una interfaz?

R: Si, no suele haber restricciones en ese sentido.

7) P: ¿Las Clases Abstractas llevan método constructor?

R: Si, todas las clases llevan constructor, incluso las abstractas, a pesar de no poder instanciarse.

8) P: Si una clase abstracta hereda de una clase común ¿las clases hijas de la clase abstracta, también podrían acceder a la clase padre de la abstracta?

R: Si, es posible, siempre y cuando los atributos y métodos de la clase padre de la abstracta sean protegidos o públicos.

9) P: ¿Es correcto el siguiente ejemplo?:

Clase publica Sansevieria implementa Botanica heredaDe Especie

¿O es necesario crear clases separadas, una que implemente la interfaz y la otra que herede la clase abstracta?

R: Una clase puede implementar entre 0 y N interfaces y (según lo que estamos viendo en este curso) una clase puede heredar de entre 0 y 1 clase.

¿Se puede combinar implementaciones de interfaces y herencia? Si, se puede. ¿En qué orden se declaran? Es indistinto, puede ser:

Clase publica Sansevieria implementa Botanica heredaDe Especie

O

Clase publica Sansevieria heredaDe Especie implementa Botanica

10) P: ¿Es posible implementar una interfaz en una clase que hereda de una clase abstracta? Por ejemplo:

clase publica Persona heredaDe Animal implementa SeresVivos

Con la "clase Animal" siendo abstracta.

R: Si, no hay restricciones en ese sentido.

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