



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



UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

**Centro de
e-Learning**

p. 2

MÓDULO 3 - UNIDAD 10

Pilares de la OO

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



Presentación:

En esta Unidad profundizamos los conceptos, siguiendo con los temas vistos anteriormente, agregando nuevos contenidos.

Se analizará uno de los principios o pilares más poderosos y que vuelve distintivo al paradigma OO: la Herencia. Este mecanismo permite la reutilización de clases, siendo este uno de los objetivos de la POO.

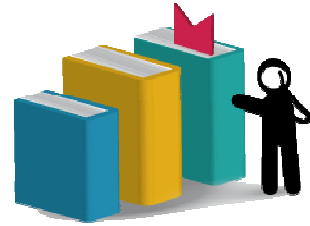
Entre otros temas, se verá también el concepto de sobrecarga de métodos, otro de los componentes distintos de la POO. Con respecto a la sobrecarga de operadores, sólo lo analizaremos conceptualmente debido a la complejidad que implica representarlo con pseudocódigo.



Objetivos:

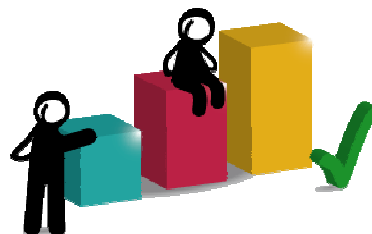
Que los participantes:

- Incorporen el concepto de modificadores de acceso para clases, métodos y atributos
- Comprendan el concepto de Herencia, uno los pilares de la POO
- Comprendan el concepto de sobrecarga, tanto de operadores como de métodos



Bloques temáticos:

1. Modificadores de acceso
2. Principios de la OO: Herencia
 - 2.1 Herencia múltiple
3. Sobrecarga de operadores
4. Sobrecarga de métodos



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. Modificadores de acceso

Retomando el concepto de encapsulamiento visto en la Unidad anterior, nos encontramos con su implementación que son los modificadores de acceso. Estos permiten al creador de las clases poner un límite a los estados o comportamientos a los cuales puede acceder otra clase que la invoque.



Los modificadores de acceso delimitan aquellas funcionalidades que son privadas y no deberían importarle o ser necesitadas a otro que no sea el mismo creador de la clase.

Por lo tanto, se puede decir que **los modificadores de acceso son la implementación concreta del encapsulamiento.**

1.1 Paquetes

Antes de continuar con los tipos de modificadores, tenemos que hacer referencia a los “**paquetes**” (o “package”, en inglés), concepto presente en la mayoría de los lenguajes de programación OO.

Un paquete es un conjunto de clases. Su uso está destinado a agrupar las clases de alguna forma, o bajo cierto criterio claro, que tenga sentido.



Comúnmente se crean paquetes:

- **Por coherencia funcional:** este tipo de agrupación hace referencia a juntar clases que tienen una afinidad a nivel de “negocio”, o sea, a partir de qué parte del problema, funcionalmente hablando, se está resolviendo con ese grupo de clases. Por ejemplo, si estamos modelando una ferretería, podríamos agrupar las clases relacionadas a las herramientas mecánicas por un lado, las herramientas manuales por otro, los artículos de electricidad por otro, etc. Cada uno de estos serían paquetes, que contendrían clases del estilo “taladro”, “amoladora” en el primer conjunto, “destornillador” y “llave inglesa” en el segundo, y “alargue” y “enchufe” en el último grupo, o paquete.
- **Por coherencia de acción:** aquí se separan las entidades modeladas según su función a nivel lógico. Por ejemplo, las clases que representan una entidad por un lado, clases que modifican entidades por otro, clases que muestran reportes de las entidades por otro, etc. Este es el tipo de uso más común en sistemas de pequeña escala.
- **Doble combinación:** es una combinación de las dos anteriores, pudiendo una contener a la otra, indistintamente. Este es el tipo de agrupación en paquetes para sistemas medianos a grandes.

Los paquetes, con los cuales se puede hacer una analogía con los directorios o carpetas de un sistema operativo, también permiten duplicar los nombres de las clases, de la misma manera que podemos tener dos planillas de Excel con el mismo nombre, en carpetas diferentes. Aunque esta práctica no es recomendable dado que se puede prestar a confusión cuál es la clase que realmente nos interesa usar o modificar, en caso de haber más de una con el mismo nombre.

Una forma posible de representar los paquetes es a través de jerarquías de directorios, donde las terminales (últimos nodos) serían las clases que se encuentran dentro de cada paquete.



Por ejemplo:

- Sistema Tienda
 - Paquete Almacén
 - ...
 - Paquete Ferretería
 - Paquete Herramientas Mecánicas
 - Clase Taladro
 - Clase Amoladora
 - Paquete Herramientas Manuales
 - Clase Destornillador
 - Clase Llave Inglesa
 - Paquete Artículos Electricidad
 - Clase Enchufe
 - Clase Alargue
 - Paquete Cajas
 - ...

De esta forma podemos ver cómo se estructura los paquetes: un sistema (una aplicación) podrá contener paquetes los cuales, a su vez, podrán contener otros paquetes y/o clases. Esto paquetes también podrán tener paquetes y/o clases y así en forma a priori indefinida.

Los paquetes se suelen identificar en la primera línea de código de cada clase, ya que es la primera información que un compilador espera encontrar.

1.1 Aplicación de los modificadores de acceso

Si bien puede haber diferencias de implementación en distintos lenguajes de programación, normalmente (en los principales lenguajes) los modificadores de acceso son aplicables a nivel de:



- **Modificadores de acceso para Clases:** suelen limitarse a dos tipos
 - Público: todas las clases ven a la clase pública
 - Privado: solo visible y accesible por las clases del mismo paquete
- **Modificadores de acceso para Atributos:**
 - Público: es posible acceder al atributo desde cualquier punto
 - Privado: solo es posible acceder al atributo desde métodos de la misma clase
 - Protegido: accesible solamente desde aquellas clases que tienen una relación de herencia (el tema de la herencia lo veremos en detalle en esta misma Unidad)
 - Sin modificador: se tiene visibilidad del atributo desde cualquier lugar del mismo paquete en donde se encuentra la clase
- **Modificadores de acceso para Métodos:**
 - Público: es posible acceder al método desde cualquier punto
 - Privado: solo es posible acceder al método desde otros métodos de la misma clase
 - Protegido: método accesible solamente desde aquellas clases que tienen una relación de herencia
 - Sin modificador: se tiene visibilidad del método desde cualquier clase mientras se trate del mismo paquete en donde se encuentra definida la clase que contiene el método

La forma de utilización de los modificadores de acceso está dada al colocar el tipo de modificador delante del nombre de la clase o método o atributo, según corresponda. La nomenclatura que utilizaremos será:



```
// ejemplo de modificador a nivel de clase
✓ clase publica MiClase

    // ejemplo de modificador a nivel de atributo
    privado String miAtributo

    // ejemplo de modificador a nivel de método
    ✓ metodo publico String miMetodo()
        miAtributo = "Holas!"
        retornar: miAtributo
    fin metodo

fin clase
```

Éstas serán las convenciones y palabras reservadas que utilizaremos en este curso, que guardan una gran similitud con los lenguajes de programación OO actuales.

Cuando se está creando una clase (o antes, idealmente) debe pensarse para qué y para quiénes se la está creando, en relación a qué atributos y métodos (sobre todo) se debería tener acceso y a cuales no; cuáles son las porciones de líneas de código que son parte interna del problema y es posible (y hasta recomendable) ocultar y a qué nivel. No todo debería ser público, ya que se agrega complejidad a la programación. Así como tampoco no todo debería ser privado, porque estaríamos creando entes aislados sin interacción, lo que tampoco tendría sentido.

Veamos los siguientes ejemplos de atributos y métodos privados y públicos.



```
✓ clase publica Alumno
    privado String nombre nuevaInstancia String()
    publico String apellido nuevaInstancia String()
    // otros atributos..

✓ metodo publico asignaNombre(String nombreNuevo)
    nombre = nombreNuevo
    fin metodo
    // otros métodos..
fin clase
```

```
clase publica CreaAlumno
    metodo publico CreaAlumno()
        Alumno alumno nuevaInstancia Alumno()

        alumno.apellido = "Medina Bello"
        alumno.asignaNombre("Ramón Ismael")

    fin metodo
fin clase
```



- “Alumno alumno nuevaInstancia Alumno()”
 - Se crea un objeto, instancia de la clase “Alumno”, llamado “alumno”
- “alumno.apellido = “Medina Bello””
 - Se accede al atributo “apellido” y se le asigna el valor “Medina Bello”. Esto es posible ya que el atributo fue definido con público.
- “alumno.asignaNombre(“Ramón Ismael”)”
 - Se envía un mensaje al método “asignaNombre” del objeto “alumno”, con el valor “Ramón Ismael” como parámetro. Esto es posible ya que el método fue definido como “publico”.
 - **No hubiera sido posible acceder al atributo “nombre” en forma directa (haciendo “alumno.nombre=“valor””), ya que fue definido como privado.**



2. Principios de la OO: Herencia

La Herencia suele ser mencionada como la prestación o característica más importante y poderosa de la orientación a objetos, ya que es uno de los medios que permiten concretar la abstracción de la que venimos hablando, tan necesaria para “modelar la realidad en objetos”. Además, provee un mecanismo para interactuar entre dichos objetos, a la vez que posibilita que sean descompuestos (o desglosados) para reducir la magnitud del problema que se está intentando solucionar, asunto sobre el cual venimos remarcando la importancia.

Concretamente, la Herencia permite generar clases a partir de otras ya existentes, extendiendo y particularizando su estado y comportamiento. Se suele hacer un paralelismo con la herencia genética entre padres e hijos. De hecho, a la clase que hereda se la suele llamar “hija” y a la otra “padre”, en un sentido jerárquico.



Con respecto a la clase padre se dice que una clase general y su hija una clase más específica, en el sentido en que la heredera toma el estado y comportamiento de la otra, aunque agregando su propio estado y comportamiento específico, de la misma forma que una persona toma la información genética de sus progenitores, a la cual le suma su propia estructura genética.

Analicemos el siguiente ejemplo:



```
clase publica Persona
    publico String nombre nuevaInstancia String()
    publico String domicilio nuevaInstancia String()
    publico Integer dni nuevaInstancia Integer()

    metodo publico asignaNombre(String nombreNuevo)
    |     nombre = nombreNuevo
    fin metodo

    metodo publico asignaDomicilio(String domicilioNuevo)
    |     domicilio = domicilioNuevo
    fin metodo

    metodo publico asignaDni(Integer DniNuevo)
    |     dni = DniNuevo
    fin metodo

    metodo publico String devuelveNombre()
    |     retornar: nombre
    fin metodo

    metodo publico String devuelveDomicilio()
    |     retornar: domicilio
    fin metodo

    metodo publico Integer devuelveDni()
    |     retornar: dni
    fin metodo

fin clase
```




```
clase publica Alumno heredaDe Persona
    privado Boolean esAlumnoRegular nuevaInstancia Boolean()
    privado Integer promedioCalificaciones nuevaInstancia Integer()

    metodo publico asignaEsAlumnoRegular(Boolean esAlumnoRegularNuevo)
        esAlumnoRegular = esAlumnoRegularNuevo
    fin metodo

    metodo publico asignaPromedioCalificaciones(Integer promedioCalificacionesNuevo)
        promedioCalificaciones = promedioCalificacionesNuevo
    fin metodo

    metodo publico Integer devuelvePromedioCalificaciones()
        retornar: promedioCalificaciones
    fin metodo

    metodo publico Boolean devuelveEsAlumnoRegular()
        retornar: esAlumnoRegular
    fin metodo

fin clase
```

```
clase publica Profesor heredaDe Persona
    privado Integer cantidadDeMaterias nuevaInstancia Integer()
    privado Float sueldo nuevaInstancia Float()

    metodo publico asignaCantidadDeMaterias(Integer cantidadDeMateriasNuevo)
        cantidadDeMaterias = cantidadDeMateriasNuevo
    fin metodo

    metodo publico asignaSueldo(Float sueldoNuevo)
        sueldo = sueldoNuevo
    fin metodo

    metodo publico Integer devuelveCantidadDeMaterias()
        retornar: cantidadDeMaterias
    fin metodo

    metodo publico Float devuelveSueldo()
        retornar: sueldo
    fin metodo

fin clase
```



```
clase publica CreoAlumnos
metodo publico CreoAlumnos()
    privado Alumno bart nuevaInstancia Alumno()
    privado Alumno lisa nuevaInstancia Alumno()
    privado Profesor edna nuevaInstancia Profesor()

    bart.asignaNombre("Bartolomeo S") //metodo que modifica atributo publico de la clase padre
    bart.asignaEsAlumnoRegular(falso) //metodo publico en Alumno, accede a atributo privado
    bart.asignaPromedioCalificaciones(2) //metodo publico en Alumno, accede a atributo privado

    lisa.asignaNombre("Lisa S") //metodo que modifica atributo publico de la clase padre
    lisa.asignaEsAlumnoRegular(verdadero) //metodo publico en Alumno, accede a atributo privado
    lisa.asignaPromedioCalificaciones(10) //metodo publico en Alumno, accede a atributo privado

    edna.asignaNombre("Edna K") //metodo que modifica atributo publico de la clase padre
    edna.asignaCantidadDeMaterias(8) //metodo publico en Profesor, accede a atributo privado

    //resto de las invocaciones

fin metodo
fin clase
```

- Introducimos aquí a nuestro diccionario una palabra reservada nueva: **“heredaDe”**. Su uso se ve reflejado en la declaración del nombre de las clases Alumno y Profesor
- Cada una de las clases van a tener el estado definido por los atributos:
 - Persona:
 - nombre
 - domicilio
 - dni
 - Alumno:
 - nombre
 - domicilio
 - dni
 - esAlumnoRegular
 - promedioCalificaciones



- Profesor:
 - nombre
 - domicilio
 - dni
 - cantidadDeMaterias
 - sueldo
- Lo mismo ocurrirá con los métodos.

Aquí se aprecia la razón de ser de la herencia:

- **Todos los alumnos son personas**
- **Todos los profesores son personas**
- **No todas las personas son profesores y alumnos, aunque algunos lo puedan ser**

De esta forma tenemos una herramienta poderosísima para crear código de calidad.

Imaginen que por un momento no pudiéramos heredar estado y/o comportamiento, lo que nos llevaría a tener los atributos y sus correspondientes métodos replicados (copiados, por triplicado) en estas tres clases.

Ahora, si por alguna resolución legal el DNI pasara de ser numérico a hexadecimal, por ejemplo, la consecuencia sería que tendríamos que modificar tres clases, en lugar de una, si tuviéramos la posibilidad de usar relaciones de herencia. Si bien este es un ejemplo algo burdo, es extremadamente común que estos cambios se produzcan en los sistemas.

Probablemente no cambie nunca el tipo de dato del DNI, aunque recordemos las siguientes situaciones:

- Cambio en la definición del dominio de los vehículos (patentes o “chapas”): pasaron de tener un formato de “letra + (número de hasta 7 cifras)” a tres letras y tres números (aunque recientemente hubo una modificación... nuevamente!)
- Cambio en el formato de los números telefónicos: con el agregado de 4 delante del numero en Capital y Gran Buenos Aires y otros formatos mucho más complejos en otras locaciones



- Y2K: el tan temido (y sobredimensionado) “efecto del año 2000”, en cual todas las fechas definidas con formas dd-mm-aa (dos dígitos para día, dos dígitos para mes, dos dígitos para año) pasó a ser dd-mm-aaaa (pasando a cuatro dígitos para el año)

En caso de haber tenido objetos que modelaran esas entidades, seguramente el costo de “refactoring” (re trabajo) hubiera sido menor. Y estamos mencionando grandes cambios, prácticamente emblemáticos. Cuando a diario nos encontramos con cambios de resoluciones del BCRA (Banco Central de la República Argentina), la AFIP, regulaciones provinciales o municipales, por nombrar algunas entidades públicas, aunque sin dejar de lado modificaciones en el ámbito privado, a través de cambios en estándares internacionales, tendencias de los mercados, el crecimiento de una organización que le lleva replantear sus procesos y sistemas, etc.

Este tipo de herencia, en la cual existe una clase padre y n clases hijas, recibe el nombre de herencia simple, para diferenciarla del caso que analizaremos a continuación. Vale aclarar que no existe un límite en cuanto a la “profundidad” del árbol de herencia (padre, hijo, nietos, bisnietos, etc., son permitidos).

2.1 Herencia múltiple

Este tipo de herencia se da cuando una clase hija tiene más de un padre. Este tipo de relación incrementan notablemente las capacidades de la herencia, al poder tener objetos modelados de una forma mucho más óptima.

Sin embargo, la herencia múltiple ha sido muy criticada con el correr de los años, debido a diferencias entre los creadores de los lenguajes que, en cierto punto, terminan siendo discusiones más filosóficas que tecnológicas. Los que rechazan su uso alegan altas posibilidades de fallas por conflictos con los nombres de atributos y métodos, lo que podría generar comportamientos no esperados o controlados.

A tal punto llegan estas discrepancias, que en la mayoría de los lenguajes de programación OO modernos (incluyendo Java, C# / .Net, Smalltalk, entre otros) se optó por incluir solamente herencia simple, aunque otros prefieren mantener la posibilidad de herencia múltiple (como C++, PHP o Eiffel).

Un breve ejemplo de herencia múltiple podría darse modelando parte de la fauna:



```
clase publica Animal
| // atributos de todos los animales ...
| // comportamiento de todos los animales ...
fin clase

clase publica Herbivoro heredaDe Animal
| // atributos los animales herbívoros...
| // métodos de los animales herbívoros ...
fin clase

clase publica Carnivoro heredaDe Animal
| // atributos los animales carnívoros ...
| // métodos de los animales carnívoros ...
fin clase

clase publica Omnivoro heredaDe Hervivoro, Carnivoro
| // atributos los animales omnívoros ...
| // métodos de los animales omnívoros ...
fin clase
```



3. Sobrecarga de operadores

Los operadores son un tipo de identificadores que indican al compilador que una porción de código va a realizar una serie de operaciones (de distinto tipo) sobre atributos u objetos.

Sobrecargar un operador implica modificar su comportamiento dado (por “default”) cuando se invoca en cierta porción de código dentro de una clase. De esta forma se reutiliza un mismo operador para usos diferentes, siendo el compilador quien decide cómo usar ese operador dependiendo sobre qué opera.

Esto es posible de hacer sobre prácticamente cualquier operador: “*”, “/”, “+”, “-”, “<<”, “>>”, “>”, “<”, etc.

Con la sobrecarga de operadores podemos, por ejemplo:

- Obtener estadísticas de las operaciones que se realizan en un programa de forma sencilla, para poder conocer el “costo” o esfuerzo por parte del conjunto de sistema operativo + software de base + hardware
- Podríamos sobrecargar el operador “+” para simplificar el algoritmo de intercalación de dos vectores, simplemente haciendo “a[] + b[]”
- O podríamos sobrecargar el operador “*” para que sepa cómo calcular una superficie multiplicando un atributo que tenga centímetros y otro que tenga metros, por dar otro ejemplo.

La forma de implementar la sobrecarga dependerá de cada lenguaje de programación. Debido a la falta de consenso general en cuanto a esto y por la complejidad que implicaría poder verificar la correctitud de un ejemplo de sobrecarga en ausencia de un compilador, evitaremos profundizar en este aspecto.

Solamente citaremos a modo de ejemplo que una sobrecarga de operador tiene similitud en cuanto a su sintaxis con la declaración de un método, con la diferencia de colocar un operador en lugar de un nombre de método.

Vale mencionar que es otro aspecto no incluido en todos los lenguajes de programación OO, como Java o Smalltalk, aunque sí es aceptado por el histórico C++ y el moderno C# (parte de la plataforma .NET).



4. Sobrecarga de métodos

La sobrecarga de métodos se refiere a la creación de varios métodos con el mismo nombre pero con distintas firmas, concepto similar a la sobrecarga de operadores.



Antes de avanzar en la definición diremos que la “firma” de un método está compuesta por los modificadores, el tipo de dato que retorna, el nombre del método y la lista de parámetros que recibe.

Por ejemplo:

- **metodo publico String realizarCalculo(Integer, Integer)**

Esta es la firma del método “realizarCalculo”.

Notar que la firma de un método es equivalente a la declaración del método, con la diferencia que se eliminan los nombres de los parámetros

Este concepto está íntimamente relacionado con otro de los grandes principios de la OO: el polimorfismo que analizaremos en la próxima Unidad.

La implementación de la sobrecarga de métodos depende de cada lenguaje de programación. A modo de ejemplo, podemos citar a Java, en el cual la sobrecarga está dada por el número de parámetros que recibe cada método y del tipo de datos de cada uno de estos parámetros, aunque no depende del tipo que devuelve.

Un ejemplo de sobrecarga de métodos podría ser:



```
class publica Cliente heredaDe Persona
  metodo publico Integer calculaSaldo(Integer saldoAnterior)
  | // ...
  fin metodo
  metodo publico Integer calculaSaldo(Integer saldoAnterior, Integer descuento)
  | // ...
  fin metodo

  metodo publico Boolean esResponsableIncripto()
  | // ...
  fin metodo
  metodo publico String esResponsableIncripto()
  | // ...
  fin metodo
fin clase
```

En este ejemplo, analizamos que:

- El método “calculaSaldo” está sobrecargado por los métodos:
 - o “metodo publico Integer calculaSaldo(Integer saldoAnterior)”
 - Este método realizaría una serie de cálculos destinados a calcular el estado de cuenta de un cliente en particular, teniendo en cuenta el saldo anterior del cliente, que se suma al saldo
 - o “metodo publico Integer calculaSaldo(Integer saldoAnterior, Integer descuento)”
 - Este método realizaría una serie de cálculos destinados a calcular el estado de cuenta de un cliente en particular, teniendo en cuenta el saldo anterior del cliente, que se suma al saldo pero se le restaría el descuento que llega también como parámetro
 - o En este caso HAY sobrecarga, ya que los modificadores de acceso son iguales, los tipos de datos devueltos son igual, el nombre de los métodos son iguales, pero la lista de parámetros (argumentos) es distinta.
- En el caso del método “esResponsableIncripto” **el compilador nos daría un error**, ya que interpretaría que no se trata de métodos sobrecargados, a pesar de tener firmas distintas. Esto se debe a que uno de los métodos devuelve un Boolean y el otro devuelve un String.



- Para que haya sobre carga, ambos métodos deberían devolver o Boolean o String, pero ambos el mismo tipo de dato.

Otro ejemplo de sobrecarga estaría dado por parte de lo visto en la Unidad anterior, cuando definimos más de un constructor para una clase. A esto se denomina sobrecarga de constructores.

Reglas para que haya (o no) sobrecarga:

- Debe existir más de un método con el mismo nombre
- Deben tener el mismo modificador de acceso
- Deben tener el mismo tipo que devuelven (o no devolver valores)
- Deben tener diferentes parámetros:
 - Diferente cantidad
 - y/o
 - Diferente tipos de datos



5. Tips de uso y ejemplos

- Sobrecarga: veamos este ejemplo de sobrecarga de constructores

```
clase publica Cartuchera
    privado Integer cantidad nuevaInstancia Integer()

    metodo publico Cartuchera()
    |    cantidad = 0
    fin metodo

    metodo publico Cartuchera(Integer cantNueva)
    |    cantidad = cantNueva
    fin metodo

    metodo publico Cartuchera(Integer cantNueva, Boolean tieneTijeras)
    |    cantidad = cantNueva
    |    si tieneTijeras = verdadero entonces
    |    |    mostrar: "Tiene tijeras!"
    |    fin si
    fin metodo

    /* ERROR!
    metodo publico Cartuchera(Integer cantidadDeLapices)
    |    mostrar: "La cantidad de lapices es: " + cantidadDeLapices
    fin metodo
    */

fin clase
```

En este caso hay 4 constructores (4 métodos con el mismo nombre), por lo que podremos tener sobrecarga o métodos repetidos. Para comprobar esto, analizaremos las firmas de los métodos, en orden:



- 1) Método constructor:

```
metodo publico Cartuchera()
```

- Firma:

- **metodo publico Cartuchera()**: este es el constructor por default (porque no tiene parámetros)

- 2) Método constructor:

```
metodo publico Cartuchera(Integer cantNueva)
```

- Firma:

- **metodo publico Cartuchera(Integer)**: este es un constructor sobrecargado, porque se diferencia del constructor por default por tener un parámetro

- 3) Método constructor:

```
metodo publico Cartuchera(Integer cantNueva, Boolean tieneTijeras)
```

- Firma:

- **metodo publico Cartuchera(Integer, Boolean)**: este es un constructor sobrecargado, porque se diferencia de los anteriores por tener diferente cantidad de parámetros.

- 4) Método constructor:

```
metodo publico Cartuchera(Integer cantidadDeLapices)
```

- Firma:

- **metodo publico Cartuchera(Integer)**: este es un método constructor repetido, ya que la firma es exactamente igual a la de un constructor anterior (el segundo), por lo tanto, no hay sobrecarga

Con los métodos de negocio (que no sean constructores) funciona de la misma manera.



No es obligatorio tener ni forzar que haya métodos de negocio con el mismo nombre. Pero si queremos aprovechar la posibilidad de tener diferentes constructores para poder crear un objeto de diferentes formas, si o si vamos a tener que implementar sobrecarga en dichos constructores.

- Herencia: veamos un ejemplo de herencia y cómo una clase (la que hereda) puede acceder a los atributos y métodos de la otra clase (la clase padre):

```
class publica TheSimpsons
    public String creador nuevaInstancia String("Matt Groening")

    metodo publico TheSimpsons()
    fin metodo

    metodo publico mensaje()
    |    mostrar: "A la grande le puse Cuca"
    fin metodo
fin clase

class publica Futurama heredaDe TheSimpsons

    metodo publico Futurama()
    |    mostrar: "El creador es "
    |    mostrar: creador          //accedo al atributo de la clase padre
    |    |    //muestra "Matt Groening"
    |
    |    creador = "Nik"          //modifico al atributo de la clase padre
    |    mostrar: "El creador es "
    |    mostrar: creador
    |    |    //muestra "Nik"
    |
    |    mensaje()              //accedo al metodo heredado como si fuera un
    |    |    // método de la clase
    fin metodo
fin clase
```



- Las clases que heredan de otra NO reemplazan a las clase de integración o test.
- Las clase de integración NO deben heredar de clases propias, ya que deben ser independientes

- Modificadores de acceso: veamos un ejemplo más:

```
clase publica Uno

    privado Integer primero nuevaInstancia Integer()
    publico Integer segundo nuevaInstancia Integer()

    metodo publico Uno()
    fin metodo

    metodo publico Integer devuelvoPrimero()
    |   retornar: primero
    fin metodo

    metodo privado Integer devuelvoSegundo()
    |   retornar: segundo
    fin metodo

fin clase

clase publica Dos
    privado Uno u nuevaInstancia Uno()

    metodo publico Dos()
    |   mostrar: u.primerio           //accedo a atributo privado ERROR!!!
    |   mostrar: u.segundo           //accedo a atributo público OK!!!
    |   mostrar: u.devuelvoPrimero()  //accedo a método público OK!!!
    |   mostrar: u.devuelvoSegundo()  //accedo a método privado ERROR!!!
    fin metodo
fin clase
```



ACTIVIDAD DE ANÁLISIS:

- Requerimiento 1: Escribir una clase "Automovil" que tenga 3 atributos (con sus modificadores de acceso) y 2 métodos (con sus modificadores de acceso, sin incluir el o los constructores ni getters y setters), que tengan sentido y coherencia con el modelado de un vehículo
- Requerimiento 2: Escribir las clases necesarias para modelar a través del uso de la herencia las siguientes entidades: "Perro", "Mamífero", "Hombre". Cada clase debe tener, al menos, 1 atributo y 1 método. Una de esas clases (1, no todas) deberán tener el constructor o algún método sobrecargado. NO agregar más clases a las ya mencionadas.

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

Previamente pueden consultar el "ANEXO 2 - Checklist de código".

Centro de e-Learning SCEU UTN - BA.

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

www.sceu.frba.utn.edu.ar/e-learning



Centro de Elearning - FRBA - UTN

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.

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



ANEXO 1 - Respuesta Actividad de Análisis

¿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

Centro de E-learning - FRBA - UTN



Página dejada intencionalmente en blanco

Centro de E-learning - FRBA - UTN



Requerimiento 1 - Ejemplo 1)

```
clase publica Automovil
    privado String partesDelAuto [20] nuevaInstancia String()
    privado String decoradoDelAuto [3] nuevaInstancia String()
    publico Integer i nuevaInstancia Integer()

    metodo publico Automovil()
        i = 1
    fin metodo

    metodo publico asignarPartesDelAuto()
        mientras i <= 20 hacer
            mostrar: "Ingrese la parte del auto número " + i
            ingresar: partesDelAuto[i]
            i = i + 1
        fin mientras
    fin metodo

    metodo publico asignarDecoracionesDelAuto()
        mostrar: "Ingrese el color del auto "
        ingresar: decoradoDelAuto [1]

        mostrar: "Ingrese el color de tapizado "
        ingresar: decoradoDelAuto [2]

        mostrar: "Ingrese SI, si los vidrios son polarizados, o NO si no lo son "
        ingresar: decoradoDelAuto [3]
    fin metodo
fin clase

clase publica AutomovilTest
    metodo publico AutomovilTest()
        Automovil a nuevaInstancia Automovil()
        a.asignarDecoracionesDelAuto()
        a.asignarPartesDelAuto()
    fin metodo
fin clase
```



Requerimiento 1 - Ejemplo 2)

```
clase publica Automovil
    privado String duenio nuevaInstancia String()
    privado String patente nuevaInstancia String()
    publico Boolean multasPendientes nuevaInstancia Boolean()

    metodo publico Automovil()
        duenio = "Ricardo Fort"
        patente = "RF111FF"
    fin metodo

    metodo publico multas(Boolean m)
        multasPendientes = m
    fin metodo

    metodo publico String mostrarTitularYPatente()
        retornar: "El titular del vehiculo:" + duenio + "Patente:" + patente
    fin metodo

    metodo publico mostrarEstadoDeMultas()
        si multasPendientes = verdadero entonces
            mostrar: "El vehiculo tiene multas sin pagar"
        sino
            mostrar: "El vehiculo no tiene multas"
        fin si
    fin metodo

    metodo publico cambiarDuenio(String nuevoDuenio)
        si nuevoDuenio = duenio entonces
            mostrar: "El nombre ingresado, es del actual titular. Ingrese al nuevo titular."
            ingresar: duenio
        sino
            mostrar: "El nuevo duenio del vehiculo es" + duenio + "."
        fin si
        duenio = nuevoDuenio
    fin metodo
fin clase
```



```
clase publica AutomovilTest
    Automovil a nuevaInstancia Automovil()

    metodo publico AutomovilTest()
        modificarMultasYMostrar(verdadero)
        modificarDuenio("Gustavo Santaolalla")
        datosFinales()
    fin metodo

    metodo publico modificarMultasYMostrar(Boolean vof)
        a.multas(vof)
        a.mostrarEstadoDeMultas()
    fin metodo

    metodo publico modificarDuenio(String nuevoDuenio)
        a.cambiarDuenio(nuevoDuenio)
    fin metodo

    metodo publico datosFinales()
        a.mostrarTitularYPatente()
    fin metodo

fin clase
```



Requerimiento 1 - Ejemplo 3)

```
clase publica Automovil

    privado String nombreModelo nuevaInstancia String()
    privado Float precioDelAutomovil nuevaInstancia Float()
    privado Float precioEnCuotas nuevaInstancia Float()

    metodo Automovil()
        ingresar: nombreModelo
        establecerPrecio()
        ahora12()
    fin metodo

    metodo privado establecerPrecio()
        en caso de nombreModelo hacer
            caso = "Aventador"
                precioDelAutomovil = 37669924.35
            fin caso
            caso = "Murcielago LP"
                precioDelAutomovil = 37614745.61
            fin caso
            caso = "Huracan EVO"
                precioDelAutomovil = 28713265.00
            fin caso
            sino
                mostrar: "modelo desconocido, reintente"
                establecerPrecio()
            fin en caso de
        fin metodo

    metodo privado ahora12()
        precioEnCuotas = precioDelAutomovil / 12,00
        mostrar: "Las cuotas para su Lamborghini " + nombreModelo + " serán de " + precioEnCuotas
    fin metodo
fin clase
```



Requerimiento 1 - Ejemplo 4)

```
clase publica Automovil
    publico String marca nuevaInstancia String()
    privado Integer año nuevaInstancia Integer()
    privado Boolean automatico nuevaInstancia Boolean()

    metodo publico Automovil()
    fin metodo

    metodo publico Integer obtenerElAño()
    |   retornar: año
    fin metodo

    metodo publico asignarElAño(Integer añoNuevo1)
    |   año = añoNuevo1
    fin metodo

    metodo publico Boolean obtenerAutomatico()
    |   retornar: automatico
    fin metodo

    metodo publico asignarAutomatico(Boolean automaticoNuevo2)
    |   automatico = automaticoNuevo2
    fin metodo

    metodo publico esNuevo()
    |   si año >= 2019 entonces
    |       mostrar: "si"
    |   sino
    |       mostrar: "no"
    |   fin si
    fin metodo
fin clase

clase publica AutomovilTest
    metodo publico AutomovilTest()
    |   Automovil a nuevaInstancia Automovil()
    |   a.asignarElAño(2018)
    |   a.asignarAutomatico(verdadero)
    |   a.marca = "Sentra"
    |   a.esNuevo()
    fin metodo
fin clase
```



Requerimiento 1 - Ejemplo 5)

```
clase publica Auto
    privado String marca nuevaInstancia String()
    privado Integer valorMercado nuevaInstancia Integer()
    privado Integer kilometraje nuevaInstancia Integer()

    metodo publico Auto()
    fin metodo

    metodo publico asignoMarca(String nuevaMarca)
    |   marca = nuevaMarca
    fin metodo

    metodo publico asignoValor(Integer valorNuevo)
    |   valorMercado = valorNuevo
    fin metodo

    metodo publico rechazoOnoAuto(Integer nuevoKM)
    |   si nuevoKM >= 200000 entonces
    |   |   mostrar: "Operación rechazada, el auto excede KM aceptable"
    |   sino
    |   |   mostrar: "Auto a la venta"
    |   fin si
    fin metodo
fin clase

clase publica AutoTest
    metodo publico AutoTest()
    |   Auto a nuevaInstancia Auto()
    |   a.asignoMarca("SIAM")
    |   a.asignoValor(100000)
    |   a.rechazoOnoAuto(27727)
    fin metodo
fin clase
```




Requerimiento 1 - Ejemplo 6)

```
clase publica Automovil
    privado String marca nuevaInstancia String()
    privado String modelo nuevaInstancia String()
    publico String check nuevaInstancia String()

    metodo publico Automovil()
        marca = ""
        modelo = ""
    fin metodo

    metodo publico String rdoMarca()
        retornar: marca
    fin metodo

    metodo publico String rdoModelo()
        retornar: modelo
    fin metodo

    metodo publico ingresarMarca(String nuevaMarca)
        marca = nuevaMarca
    fin metodo

    metodo publico ingresarModelo(String nuevoModelo)
        modelo = nuevoModelo
    fin metodo

    metodo publico check()
        check = marca + modelo
    fin metodo

    metodo publico controlCheck()
        si check = "Chevrolet Cruze" entonces
            mostrar: "Correcto"
        sino
            mostrar: "Incorrecto"
        fin si
    fin metodo
fin clase

clase publica AutomovilTest
    metodo publico AutomovilTest()
        Automovil a nuevaInstancia Automovil()
        a.ingresarMarca("Chevrolet")
        a.ingresarModelo("Cruze")
        a.check()
        a.controlCheck()
        mostrar: "la marca y modelo son " + a.check
    fin metodo
fin clase
```



Requerimiento 1 - Ejemplo 7)

```
clase publica Automovil
    privado String marca nuevaInstancia String()
    privado String modelo nuevaInstancia String()

    metodo publico Automovil()
        marca = ""
        modelo = ""
    fin metodo

    metodo publico asignarMarca (String setMarca)
        marca = setMarca
    fin metodo

    metodo publico asignarModelo (String setModelo)
        modelo = setModelo
    fin metodo

    metodo privado String obtenerMarca()
        retornar: marca
    fin metodo

    metodo privado String obtenerModelo()
        retornar: modelo
    fin metodo

    metodo publico verMarca()
        si marca != "" entonces
            mostrar: "Usted ha ingresado la marca " + obtenerMarca()
        sino
            mostrar: "Usted no ha ingresado ninguna marca aún"
        fin si
    fin metodo

    metodo publico verModelo()
        si modelo != "" entonces
            mostrar: "Usted ha ingresado el modelo " + obtenerModelo()
        sino
            mostrar: "Usted no ha ingresado ningún modelo aún"
        fin si
    fin metodo
fin clase
```



```
clase publica AutomovilTest
    metodo publico AutomovilTest()
        Automovil a nuevaInstancia Automovil()
        a.asignarMarca("SIAM")
        a.asignarModelo("Gordini")
        a.verMarca()
        a.verModelo()
    fin metodo
fin clase
```



Requerimiento 1 - Ejemplo 8)

```
clase publica Automovil
    privado String marca nuevaInstancia String()
    privado String color nuevaInstancia String()
    privado Integer puertas nuevaInstancia Integer()

    metodo publico setMarca(String nuevaMarca)
    |   marca = nuevaMarca
    fin metodo

    metodo publico String getMarca()
    |   retornar: marca
    fin metodo

    metodo publico setColor(String nuevoColor)
    |   color = nuevoColor
    fin metodo

    metodo publico String getColor()
    |   retornar: color
    fin metodo

    metodo publico setPuertas(Integer cantPuertas)
    |   puertas = cantPuertas
    fin metodo

    metodo publico Integer getPuertas()
    |   retornar: puertas
    fin metodo

    metodo publico Automovil()
    |   setMarca("")
    |   setColor("")
    |   setPuertas(0)
    fin metodo

    metodo publico Automovil(String marca1, String color1, Integer puertas1)
    |   setMarca(marca1)
    |   setColor(color1)
    |   setPuertas(puertas1)
    fin metodo
```



```
metodo publico Boolean esCoupe()
    si getPuertas() > 2 entonces
        retornar: falso
    sino
        retornar: verdadero
    fin si
fin metodo

metodo publico Boolean fabricadoEnArg()
    si getMarca() = "chevrolet" O getMarca() = "citroen" O getMarca() = "fiat" entonces
        retornar: verdadero
    sino
        retornar: falso
    fin si
fin metodo
fin clase

clase publica AutomovilTest
    metodo publico AutomovilTest()
        Automovil a nuevaInstancia Automovil("fiat", "gris", 5)
        mostrar: "verificamos si es coupe "
        si a.esCoupe() = verdadero entonces
            mostrar: "si, lo es"
        sino
            mostrar: "no, no lo es"
        fin si
        mostrar: "verificamos si es de producción argentina "
        mostrar: a.fabricadoEnArg()
    fin metodo
fin clase
```



Requerimiento 1 - Ejemplo 9)

```
clase publica Automovil
    privado String color nuevaInstancia String()
    privado String clasificacion nuevaInstancia String()
    privado String marca nuevaInstancia String()

    metodo publico Automovil ()
    fin metodo

    metodo publico Automovil (String nColor, String nClasificacion, String nMarca)
    |   color = nColor
    |   clasificacion = nClasificacion
    |   marca = nMarca
    fin metodo

    metodo publico definirColor(String nColor)
    |   color = nColor
    fin metodo

    metodo publico definirClasificacion(String nClasificacion)
    |   clasificacion = nClasificacion
    fin metodo

    metodo publico definirMarca(String nMarca)
    |   marca = nMarca
    fin metodo

    metodo publico String darColor()
    |   retornar: color
    fin metodo

    metodo publico String darClasificacion()
    |   retornar: clasificacion
    fin metodo

    metodo publico String darMarca()
    |   retornar: marca
    fin metodo
```



```
metodo publico verificarMarca(String nMarca)
    si nMarca = marca entonces
        mostrar: "La marca del Automovil coincide y es: " + marca
    sino
        mostrar: "La marca del automovil no coincide"
    fin si
fin metodo

metodo publico Boolean verificarClasificacion(String nClasificacion)
    Boolean ver nuevaInstancia Boolean(falso)

    si nClasificacion != "" entonces
        ver = verdadero
    fin si

    retornar: ver
fin metodo

fin clase

clase publica AutomovilTest
    metodo publico AutomovilTest()
        Automovil a nuevaInstancia Automovil()
        a.verificarMarca("Ford")
        mostrar: a.verificarClasificacion("ABC")
    fin metodo
fin clase
```



Requerimiento 1 - Ejemplo 10)

```
clase publica Automovil

    publico String ubicacion nuevaInstancia String()
    privado String marca nuevaInstancia String()
    privado String modelo nuevaInstancia String ()

    metodo publico Automovil()
    |   ubicacion = "La Plata"
    fin metodo

    metodo publico asignarUbicacion(String nuevaUbicacion)
    |   ubicacion = nuevaUbicacion
    fin metodo

    metodo publico String obtenerUbicacion()
    |   retornar: ubicacion
    fin metodo

    metodo publico asignarMarca(String nuevaMarca)
    |   marca = nuevaMarca
    fin metodo

    metodo publico String obtenerMarca()
    |   retornar: marca
    fin metodo

    metodo publico asignarModelo(String nuevoModelo)
    |   modelo = nuevoModelo
    fin metodo

    metodo publico String obtenerModelo()
    |   retornar: modelo
    fin metodo

    metodo publico String filtrarUbicacion()
    |   ubicacion = ""
    |   mostrar: "ingresar la nueva ubicacion "
    |   ingresar: ubicacion
    |   mostrar: "nueva ubicacion "
    |   retornar: ubicacion
    fin metodo
fin clase
```




```
clase publica AutomovilTest
metodo publico AutomovilTest()
    Automovil a nuevaInstancia Automovil()
    a.asignarMarca("Ford")
    a.asignarModelo("Ranger")
    mostrar: a.filtrarUbicacion()
fin metodo
fin clase
```



Requerimiento 2 - Ejemplo 1)

```
clase publica Hombre heredaDe Mamifero
    metodo publico Hombre()
    |    mostrar: "soy un hombre"
    fin metodo

fin clase

clase publica Mamifero
    publica String nombre nuevaInstancia String()

    metodo publico Mamifero()
    |    nombre = ""
    fin metodo

    metodo publico darNombre(String iNombre)
    |    nombre = iNombre
    fin metodo

    metodo publico darNombre()
    |    nombre = "Ruperto Pesto"
    fin metodo
fin clase

clase publica Perro heredaDe Mamifero
    metodo publico Perro()
    |    nombre = ""
    fin metodo
fin clase

clase publica Test
    metodo publico Test()
    |    Hombre h nuevaInstancia Hombre()
    |    h.darNombre()
    |    mostrar: h.nombre
    |    Perro p nuevaInstancia Perro()
    |    p.darNombre("Firula")
    |    mostrar: p.nombre
    fin metodo
fin clase
```



Requerimiento 2 - Ejemplo 2)

```
clase publica Mamifero
    protegido String pelo nuevaInstancia String()

    metodo publico Mamifero()
    fin metodo

    metodo publico Mamifero(String elPelo)
    |    pelo = elPelo
    fin metodo

    metodo protegido mostrarPelo()
    |    si pelo != "" entonces
    |    |    mostrar: pelo
    |    sino
    |    |    mostrar: "sin datos"
    |    fin si
    fin metodo
fin clase

clase publica Hombre heredaDe Mamifero
    metodo publico Hombre()
    fin metodo

    metodo publico descubrirPelo()
    |    pelo = "este hombre tiene el pelo que cambia periodicamente"
    fin metodo
fin clase

clase publica Perro heredaDe Mamifero
    publico String colorOjos nuevaInstancia String()

    metodo publico Perro()
    |    colorOjos = "marrones"
    fin metodo
fin clase

clase publica Test
    metodo publico Test()
    |    Hombre h nuevaInstancia Hombre()
    |    h.mostrarPelo()
    |    mostrar: h.descubrirPelo()
    |    Perro p nuevaInstancia Perro()
    |    p.mostrarPelo()
    |    mostrar: p.colorOjos
    fin metodo
fin clase
```



Requerimiento 2 - Ejemplo 3)

```
class publica Mamiferos
    publico String claseAnimal nuevaInstancia String()
    publico String claseMamifero nuevaInstancia String()

    metodo publico Mamifero()
    fin metodo

    metodo protegido clase()
    |   claseAnimal = "vertebrado"
    fin metodo

    metodo protegido clase(String tipo)
    |   claseAnimal = tipo
    fin metodo
fin clase

class publica Hombre heredaDe Mamifero
    metodo publico Hombre()
    |   claseMamifero = "bipedo"
    |   clase() //invoco al método de la clase padre
    fin metodo
fin clase

class publica Perro heredaDe Mamifero
    metodo publico Perro()
    |   claseMamifero = "cuadrupedo"
    |   clase() //invoco al método de la clase padre
    fin metodo
fin clase

class publica Test
    metodo pu Test()
    |   Hombre h nuevaInstancia Hombre()
    |   mostrar: h.claseAnimal
    |   mostrar: h.claseMamifero
    |   Perro p nuevaInstancia Perro()
    |   mostrar: p.claseAnimal
    |   mostrar: p.claseMamifero
    fin metodo
fin clase
```



Requerimiento 2 - Ejemplo 4)

```
clase publica Mamifero
    publico String alimentacion nuevaInstancia String()

    metodo publico Mamifero()
        alimentacion = ""
    fin metodo

    metodo publico asignaAlimentacion(String nAlimentacion)
        alimentacion = nAlimentacion
    fin metodo
fin clase

clase publica Perro heredaDe Mamifero
    publico String raza nuevaInstancia String()

    metodo publico Perro()
        raza = ""
        asignaAlimentacion("omnivoros")
    fin metodo

    metodo publico asignaRaza(String nRaza)
        raza = nRaza
    fin metodo
fin clase

clase publica Hombre heredaDe Mamifero
    publico String genero nuevaInstancia String()

    metodo publico Hombre()
        genero = ""
        asignaAlimentacion("omnivoros")
    fin metodo

    metodo publico Hombre(String nGenero)
        genero = nGenero
        asignaAlimentacion("omnivoros")
    fin metodo

    metodo publico asignaNutricion(String nutricion)
        asignaAlimentacion(nutricion)
    fin metodo
fin clase
```



```
clase publica Test
metodo publico Test()
    Hombre h nuevaInstancia Hombre("N/A")
    h.asignaNutricion("vegano")
    Perro p nuevaInstancia Perro()
    p.asignaRaza("callejero")
fin metodo
fin clase
```



Requerimiento 2 - Ejemplo 5)

```
clase publica Mamifero
    protegido String clasificacion nuevaInstancia String()

    metodo publico Mamifero(String nClasificacion)
    |   clasificacion = nClasificacion
    fin metodo

    metodo protegido String darClasificacion()
    |   retornar: clasificacion
    fin metodo

    metodo protegido definirClasificacion(String nClasificacion)
    |   clasificacion = nClasificacion
    fin metodo
fin clase

clase publica Perro heredaDe Mamifero
    privado String raza nuevaInstancia String()

    metodo Perro ()
    |   raza = ""
    fin metodo

    metodo publico asignarRaza(String nRaza)
    |   raza = nRaza
    fin metodo

    metodo publico asignarRaza(String nRaza, String nClasificacion)
    |   raza = nRaza
    |   clasificacion = nClasificacion
    fin metodo

    metodo publico mostrarPerro()
    |   mostrar: "Su raza es: " + raza
    fin metodo
fin clase
```



```
clase publica Hombre heredaDe Mamifero
    privado String genero nuevaInstancia String()

    metodo publico Hombre (String nGenero)
        genero = nGenero
    fin metodo

    metodo publico Hombre (String nGenero, String nClasificacion)
        genero = nGenero
        clasificacion = nClasificacion
    fin metodo
fin clase

clase publica Test
    metodo publico Test()
        Hombre h nuevaInstancia Hombre("", "")
        h.definirClasificacion("humano")
        mostrar: h.darClasificacion()
        Perro p nuevaInstancia Perro()
        h.definirClasificacion("guau guau")
        p.asignarRaza("setter irlandés", "inquieto")
        p.mostrarPerro()
    fin metodo
fin clase
```




ANEXO 2 - Checklist de código

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

Tema	Cumple	
	Si	No
La clase y todos los métodos y atributos a nivel de clase tienen su modificador de acceso		
Los modificadores de acceso utilizados son "publico/a", "privado/a", "protegido/a"		
En una relación de herencia, los métodos/atributos de la clase padre que van a ser heredados por la clase hija son protegidos?		
En una relación de herencia, las clases hijas tienen el modificador "heredaDe" y el nombre de la clase padre en su declaración?		
Una clase hereda únicamente de una sola clase		
Si hay métodos (incluidos constructores) que tienen el mismo nombre: tiene además el mismo modificador de acceso, el mismo tipo devuelto (o no devuelven nada) y diferencias en cuanto a la cantidad y/o tipo de dato de los parámetros?		
Un objeto que invoca un método, tiene ese método declarado como "publico"?		
Un objeto que usa un atributo, tiene ese atributo declarado como "publico"?		
Se verificó que los atributos sean mayormente privados y que existan setters y getters públicos para esos atributos?		

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



ANEXO 3 - Preguntas frecuentes

1) *Pregunta: Sobre los modificadores de acceso: ¿una vez que una clase o un método es determinado de manera privada, todo lo de su interior también lo va a ser? O ¿depende de si se establece cada cosa como privada o pública?*

Respuesta: Un método privado de un objeto es "invisible" para otro objeto, por lo tanto es todo o nada. Con los atributos es más directo, porque no "contienen" nada: se ve o no se ven. Si una clase es privada, todo en su interior es privado, incluidos los constructores, por lo tanto si siquiera se puede instanciar.

2) *P: En el paradigma estructurado el programa se ejecutaba de arriba a abajo haciendo saltos, con las funciones o haciendo bucles con los "mientras". En este paradigma, ¿cómo se podría pensar que se ejecuta? ¿Se llama a una clase y desde ahí se va ejecutando el resto?*

R: Para eso necesitamos las clases de test, porque si no es un conjunto de partes que no se sabe cómo se usan ni para qué sirven. El flujo de ejecución principal nos lo va a dar la clase de test en nuestro caso.

3) *P: En cuanto a los "paquetes" ¿cómo sería la agrupación de clases por coherencia de acciones?*

R: Un paquete, físicamente, es un directorio o carpeta. La división más común de clases en una aplicación orientada a objetos suele estar relacionada con "para qué sirve" esa clase.

Por ejemplo, se podría tener:

(1) Grupo de clases que representan entidades (persona, cuenta, sucursal, alumno, figura geométrica, marca, color, etc.).

(2) Grupo de clases que se usan para guardar los estados de los objetos en una base de datos.



(2.1) Una serie de clases genéricas para comunicarte con un motor de base de datos en particular.

(3) Grupo de clases que se usan para hacer cálculos u operaciones (usando las entidades antes nombradas, por ejemplo).

Entonces podrían existir los paquetes:

nn/proyectos/miProyecto/entidades: (1)

nn/proyectos/miProyecto/accesoADatos: (2)

nn/proyectos/miProyecto/accesoADatos/utiles: (2.1)

nn/proyectos/miProyecto/servicios: (3)

4) *P: "Las clase de integración NO deben heredar de clases propias, ya que deben ser independientes". ¿A qué se refiere el concepto de "clase propia" vs "clase independiente"?*

R: "clase propia": cualquier clase que escribe el programador y no es parte del pseudocódigo (como lo son String, Date, Float, etc).

"clase independiente": relacionado con el apartado anterior, una clase que no forma parte de la jerarquía de clases atada a relación de herencia, o sea "una clase que no hereda de ninguna clase propia".

5) *P: ¿Es correcto decir que hay sobrecarga de método si hay varios métodos con el mismo nombre siempre que tengan diferente firma?*

R: Si, pero la diferencia en la firma tiene que estar dada en el tipo o cantidad de parámetros que recibe. Con esa diferencia de logra la sobrecarga.

6) *P: ¿Cuál sería el sentido y la utilidad de sobrecargar un constructor o un método?*



R: En el caso en que sea necesario o conveniente contar con diferentes constructores en una clase, para poder crear las instancias de diferente forma, no hay otra opción que que sobrecargar los constructores.

Con los métodos pasa lo mismo, pero ya es más bien una decisión del programador, pero no hay obligación ni una situación en la que haya que aplicar la sobrecarga si o si. Es un recurso disponible que se puede usar o no.

7) P: ¿Puede una clase heredar de otra clase que ya haya heredado?

R: Si, puede, dado que "no existe un límite en cuanto a la "profundidad" del árbol de herencia (padre, hijo, nietos, bisnietos, etc., son permitidos)."

Por ejemplo, en Java, todas tiene como primera clase en el árbol de herencia, al principio de todo, a la clase Object. Es la clase padre de todas las clases que existen y existirán en Java. Las que existirán son las que programarán los programadores de Java de todo el mundo (aunque, no es necesario poner "extends Object" (*) en las clases que hace uno, porque es algo que ya viene dado por el lenguaje.

(*) "extends" es una palabra reservada se usa en Java para indicar la herencia en una clase. Es nuestro "heredaDe".

8) P: ¿Qué sería un modulo de Estructurado en POO?

R: El concepto de módulo se puede aplicar a diferentes niveles o componentes en OO. Puede ser un conjunto de clases, una clase o un método, yendo desde afuera hacia adentro. En todo caso, el contexto es el que indica a qué se refiere con "módulo" cuando hablamos de OO.



Lo que vimos

- Concepto paquetes y modificadores de acceso para clases, métodos y atributos
- Concepto de Herencia, uno los pilares de la POO
- Concepto de sobrecarga, tanto de operadores como de métodos



Lo que viene:

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

