UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

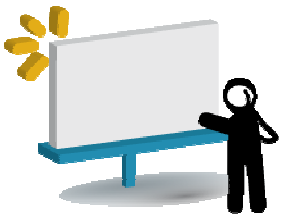
Centro de e-Learning

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 10

Pilares de la OO



Presentación:

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

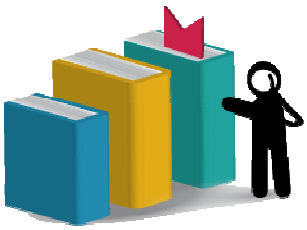
Se analizará uno de los principios o pilares más poderosos y que vuelve distintivo al paradigma OO: la Herencia. Este mecanismo permite la reutilización de clases, siendo este uno los objetivos de la POO.

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:

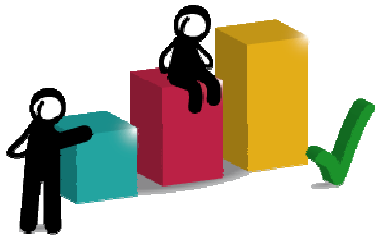
Que los participantes:

- 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:

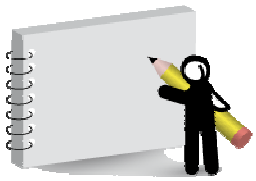
- Los foros proactivos asociados a cada una de las unidades.
- La Web 2.0.
- Los contextos de desempeño de los participantes.

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 hablado, 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 que 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 respeto 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 Alumno, 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 cases 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:

```
clase 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:
  - “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
  - “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
  - 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.

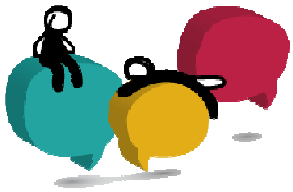
- Herencia: veamos un ejemplo de herencia y cómo una clase (la que hereda) puede acceder a los atributos y métodos de la otra clase (la clase padre):

```
clase publica TheSimpsons
    publico 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

clase 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.primero      //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 final de la Unidad 10

- Punto 1: Escribir una clase “Automovil” que tenga 3 atributos (1 de ellos público) y 2 métodos (todos públicos, sin incluir el o los constructores ni getters y setters), que tengan sentido y coherencia con el modelado de un vehículo

- Punto 2: Escribir las clases necesarias para modelar a través del uso de la herencia las siguientes entidades: “Perro”, “Mamífero”, “Hombre”. Cada clase debe tener 1 atributo y 1 método. Una de esas clases (1, no todas) deberán tener el constructor sobrecargado y 2 métodos sobrecargados. NO agregar más clases a las ya mencionadas.

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.