Contenido obligatorio Eje temático 1

Sitio:	Instituto Superior Politécnico Córdoba	Imprimido por:	Marco Virinni
Curso:	Programador Full Stack - TSDWAD - 2023	Día:	miércoles, 16 agosto 2023, 5:03 PM
Libro:	Contenido obligatorio Eje temático 1		

Descripción

Tabla de contenidos

1. Introducción

Empecemos con una introducción básica a la **P**rogramación **O**rientada a **O**bjetos **POO** o **OOP** en inglés. Se trata de un paradigma de programación introducido en los años '70, pero que no se hizo popular hasta los '90.

Este modo o paradigma de programación nos permite organizar el código de una manera que se asemeja bastante a como pensamos en la vida real, utilizando las famosas **clases**. Estas nos permiten agrupar un conjunto de variables y funciones que veremos a continuación.

Cosas de lo más cotidianas como un perro o un coche pueden ser representadas con clases. Estas clases tienen diferentes características, que en el caso del perro podrían ser la edad, el nombre o la raza. Llamaremos a estas características, **atributos**.

Por otro lado, las clases tienen un conjunto de funcionalidades o cosas que pueden hacer. En el caso del perro podría ser andar o ladrar. Llamaremos a estas funcionalidades **métodos**.

Por último, pueden existir diferentes tipos de perro. Podemos tener uno que se llama Toby o el del vecino que se llama Laika. Llamaremos a estos diferentes tipos de perro **objetos**. Es decir, el concepto abstracto de perro es la **clase**, pero Toby o cualquier otro perro particular será el **objeto**.

1.1. Motivación de la POO

Una vez explicada la programación orientada a objetos puede parecer bastante lógica, pero no es algo que haya existido siempre, y de hecho hay muchos lenguajes de programación que no la soportan.

En parte surgió debido a la creciente complejidad a la que los programadores se iban enfrentando conforme pasaban los años. En el mundo de la programación hay gran cantidad de aplicaciones que realizan tareas muy similares y es importante identificar los patrones que nos permiten no reinventar la rueda. La programación orientada a objetos intentaba resolver esto.

Uno de los primeros mecanismos que se crearon fueron las **funciones**, que permiten agrupar bloques de código que hacen una tarea específica bajo un nombre. Algo muy útil ya que permite también reusar esos módulos o funciones sin tener que copiar todo el código, tan solo la llamada.

Las funciones resultaron muy útiles, pero no eran capaces de satisfacer todas las necesidades de los programadores. Uno de los problemas de las funciones es que sólo realizan unas operaciones con unos datos de entrada para entregar una salida, pero no les importa demasiado conservar esos datos o mantener algún tipo de estado. Salvo que se devuelva un valor en la llamada a la función o se usen variables globales, todo lo que pasa dentro de la función queda olvidado, y esto en muchos casos es un problema.

Imaginemos que tenemos un juego con naves espaciales moviéndose por la pantalla. Cada nave tendrá una posición (x,y) y otros parámetros como el tipo de nave, su color o tamaño. Sin hacer uso de clases y POO, tendremos que tener una variable para cada dato que queremos almacenar: coordenadas, color, tamaño, tipo. El problema viene si tenemos 10 naves, ya que nos podríamos juntar con un número muy elevado de variables. Todo un desastre.

En el mundo de la programación existen tareas muy similares al ejemplo con las naves, y en respuesta a ello surgió la programación orientada a objetos. Una herramienta perfecta que permite resolver cierto tipo de problemas de una forma mucho más sencilla, con menos código y más organizado. Agrupa bajo una clase un conjunto de variables y funciones, que pueden ser reutilizadas con características particulares creando objetos.

2. Modularidad y Validaciones

Modularidad:

La modularidad es una forma de organización, reutilización y segmentación del código.

Los mismos son archivos independientes que se encontrarán en la carpeta raíz de la solución (o repo) y serán de extensión .py que es la que se utiliza para archivos de Python.

En lo que respecta a reutilización, se podrán desarrollar bloques de código con fines específicos para luego poder llamarlos desde cualquier parte de la solución, pudiendo ahorrar mucho tiempo de desarrollo y segmentando la lógica en pequeñas partes, lo que también ayuda en una mejor solución a la situación por la que estemos trabajando.

Otros beneficios de la modularidad son:

- ***Mantenimiento más sencillo:*** No es lo mismo encontrar un error en un código de miles de líneas, que en uno mucho más acotado y específico.
- Actualización: Si en un futuro se quiere sumar, restar o actualizar la funcionalidad, objetivo, o necesidad, tenerlo segmentado en pequeños módulos nos ofrecerá un costo menor en entender lo previo y poder enriquecerlo.
- Nos permite construir una aplicación/solución como un rompecabezas con el que voy a formar un todo, de acuerdo a la necesidad actual.

¿Cómo creo entonces un módulo?

Es muy sencillo, solo basta con ir a:

1. Archivo.
2. Nuevo archivo.
3. Ponerle un nombre y escribir el cuerpo del módulo.
4. Guardar cambios, añadiendo la extensión '.py' al nombre del archivo.

¿Cómo utilizar un módulo?

También es muy sencillo: Desde donde lo necesitamos, debemos utilizar la siguiente instrucción:

Import 'Nombre_del_modulo'

Luego, ya podremos utilizar todo lo que el módulo esté preparado para realizar.

2.1. Validaciones

Las ***VALIDACIONES*** son formas de corroborar los errores más comunes en la ejecución del código.

Las más comunes son: ***errores de sintaxis y excepciones.***

Los errores de sintaxis, también conocidos como errores de análisis, son quizás el tipo de queja más común que recibe mientras aún está aprendiendo Python:

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

El analizador repite la línea infractora y muestra una pequeña 'flecha' que apunta al primer punto de la línea donde se detectó el error. El error es causado por (o al menos detectado en) el token *que precede* a la flecha: en el ejemplo, el error se detecta en la función , ya que faltan `print()`dos puntos antes. `':'`El nombre del archivo y el número de línea se imprimen para que sepa dónde buscar en caso de que la entrada provenga de un script.

Incluso si una declaración o expresión es sintácticamente correcta, puede causar un error cuando se intenta ejecutarla. Los errores detectados durante la ejecución se denominan *excepciones* y no son incondicionalmente fatales: pronto aprenderá a manejarlos en los programas de Python. Sin embargo, la mayoría de las excepciones no son manejadas por programas y dan como resultado mensajes de error como se muestra aquí:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

La última línea del mensaje de error indica lo que sucedió. Las excepciones vienen en diferentes tipos, y el tipo se imprime como parte del mensaje: los tipos en el ejemplo son `ZeroDivisionError`, `NameError`y `TypeError`. La cadena impresa como tipo de excepción es el nombre de la excepción integrada que ocurrió. Esto es cierto para todas las excepciones integradas, pero no es necesario que lo sea para las excepciones definidas por el usuario (aunque es una convención útil). Los nombres de excepción estándar son identificadores integrados (no palabras clave reservadas).

El resto de la línea proporciona detalles según el tipo de excepción y su causa.

La parte anterior del mensaje de error muestra el contexto en el que se produjo la excepción, en forma de seguimiento de pila. En general, contiene un seguimiento de la pila que enumera las líneas de origen; sin embargo, no mostrará las líneas leídas desde la entrada estándar.

2.2. Manejo de Excepciones

Es posible escribir programas que manejen excepciones seleccionadas. Mire el siguiente ejemplo, que le pide al usuario que ingrese hasta que se haya ingresado un número entero válido, pero le permite al usuario interrumpir el programa (usando o lo que admita el sistema operativo); tenga en cuenta que una interrupción generada por el usuario se señala al generar la excepción.

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops!  That was no valid number.  Try again...")
...
```

La `try`declaración funciona de la siguiente manera.

- Primero, se ejecuta la *cláusula try* (la(s) declaración(es) entre las palabras clave `try`y).`except`
- Si no se produce ninguna excepción, se omite la *cláusula de excepción try* y finaliza la ejecución de la declaración.
- Si se produce una excepción durante la ejecución de la try cláusula, se omite el resto de la cláusula. Luego, si su tipo coincide con la excepción nombrada después de la `except` palabra clave, se ejecuta la *cláusula de excepción* y luego la ejecución continúa después del bloque try/except.
- Si ocurre una excepción que no coincide con la excepción nombrada en la *cláusula de excepción* , se pasa a las try declaraciones externas; si no se encuentra un controlador, es una *excepción no controlada* y la ejecución se detiene con un mensaje como se muestra arriba.

Una try declaración puede tener más de una *cláusula excepto* , para especificar controladores para diferentes excepciones. Como máximo se ejecutará un controlador. Los controladores solo manejan las excepciones que ocurren en la *cláusula try* correspondiente , no en otros controladores de la misma `try` declaración. Una *cláusula de excepción* puede nombrar múltiples excepciones como una tupla entre paréntesis, por ejemplo:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

Una clase en una `except`cláusula es compatible con una excepción si es la misma clase o una clase base de la misma (pero no al revés: una *cláusula de excepción que* enumera una clase derivada no es compatible con una clase base). Por ejemplo, el siguiente código imprimirá B, C, D en ese orden:

```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

La declaración `try`... tiene una *cláusula else*`except` opcional que, cuando está presente, debe seguir a todas las *cláusulas excepto* . Es útil para el código que debe ejecutarse si la *cláusula de prueba* no genera una excepción. Por ejemplo:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

El uso de la `else`cláusula es mejor que agregar código adicional a la `try`cláusula porque evita detectar accidentalmente una excepción que no fue generada por el código protegido por la instrucción `try`... `except`.

Cuando ocurre una excepción, puede tener un valor asociado, también conocido como *argumento* de la excepción . La presencia y el tipo del argumento dependen del tipo de excepción.

La *cláusula de excepción* puede especificar una variable después del nombre de la excepción. La variable está vinculada a una instancia de excepción con los argumentos almacenados en `instance.args`. Para mayor comodidad, la instancia de excepción se define `__str__()`para que los argumentos se puedan imprimir directamente sin tener que hacer referencia a `.args`. También se puede crear una instancia de una excepción antes de generarla y agregarle los atributos que desee.

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception instance
...     print(inst.args)     # arguments stored in .args
...     print(inst)          # __str__ allows args to be printed directly,
...                          # but may be overridden in exception subclasses
...     x, y = inst.args     # unpack args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

2.3. Ejercicio práctico Módulo y Validación

Retomar el ejercicio de 'Productos y Pedidos' y agregar las siguientes validaciones:

- Corroborar que al pedido se agreguen al menos dos productos.
- Corroborar que el precio de todos los productos sea mayor a cero y que sea numérico.

Link al ejercicio de productos y pedidos:

https://drive.google.com/file/d/1kxKhs8dg5v0bq4t3kuuYfb8vEZ2KWsNi/view?usp=sharing

3. Clases y Objetos

Una clase es la definición de las propiedades y comportamiento de un tipo de objeto concreto.
Es un modelo que se utiliza para describir uno o más objetos del mismo tipo.

En Python se crean de la siguiente manera:

```
# Creando una clase vacía
class Perro:
    pass
```

Se trata de una clase vacía y sin mucha utilidad práctica, pero es la mínima clase que podemos crear. Nótese el uso del `pass` que no hace realmente nada, pero daría un error si después de los `:` no tenemos contenido.

Ahora que tenemos la **clase**, podemos crear un **objeto** de la misma. Podemos hacerlo como si de una variable normal se tratase. Nombre de la variable igual a la clase con `()`. Dentro de los paréntesis irían los parámetros de entrada si los hubiera.

```
# Creamos un objeto de la clase perro
mi_perro = Perro()
```

3.1. Ejercicio práctico de clases

Crear un programa que permita crear la clase Motovechiculo, capaz de procesar la información básica de las motos en un concesionario llamado "Córdoba Motos", para ello vamos a suponer que los atributos de una moto son:

•Marca

•Modelo

•Patente

•kilometraje

Se deberá poder ingresar dichos datos y ser mostrados por consola.

Le dejamos la solución del ejercicio dado, pero se les recomienda intentar varias veces hasta que salga la solución antes de ver la misma.

Link a la resolución del ejercicio propuesto:

https://drive.google.com/file/d/1TJar-Q3f_wJpAxlNDBPqcNHipoN_H-Lr/view?usp=sharing

4. Métodos y Atributos

Los atributos son características individuales que diferencian un objeto de otro y que determinan su apariencia, estado u otras cualidades.

Los atributos se guardan en variables denominadas de instancia y cada objeto particular puede tener valores distintos para estas variables.

A continuación vamos a añadir algunos atributos a nuestra clase. Antes de nada es importante distinguir que existen dos tipos de atributos:

- Atributos de **instancia**: Pertenecen a la instancia de la clase o al objeto. Son atributos particulares de cada instancia, en nuestro caso de cada perro.
- Atributos de **clase**: Se trata de atributos que pertenecen a la clase, por lo tanto serán comunes para todos los objetos.

Empecemos creando un par de **atributos de instancia** para nuestro perro, el `nombre` y la `raza`. Para ello creamos un método `__init__` que será llamado automáticamente cuando creemos un objeto. Se trata del **constructor**.

```
class Perro:
    # El método __init__ es llamado al crear el objeto
    def __init__(self, nombre, raza):
        print(f"Creando perro {nombre}, {raza}")

        # Atributos de instancia
        self.nombre = nombre
        self.raza = raza
```

Ahora que hemos definido el método *init* con dos parámetros de entrada, podemos crear el objeto pasando el valor de los atributos. Usando `type()` podemos ver como efectivamente el objeto es de la clase `Perro`.

```
mi_perro = Perro("Toby", "Bulldog")
print(type(mi_perro))
# Creando perro Toby, Bulldog
# <class '__main__.Perro'>
```

Seguramente te hayas fijado en el `self` que se pasa como parámetro de entrada del método. Es una variable que representa la instancia de la clase, y deberá estar siempre ahí.

El uso de `__init__` y el doble `__` no es una coincidencia. Cuando veas un método con esa forma, significa que está reservado para un uso especial del lenguaje. En este caso sería lo que se conoce como **constructor**. Hay gente que llama a estos métodos mágicos.

Por último, podemos acceder a los atributos usando el objeto y `.`. Por lo tanto.

```
print(mi_perro.nombre) # Toby
print(mi_perro.raza)   # Bulldog
```

4.1. Ejercicio práctico atributos

Luego del análisis de la clase de Motovehículo, nos hemos dado cuenta que además de las info que se guarda en dicha clase, necesitamos agregar:
•Año de patentamiento - SOLO AÑO.
•Color.
•Capacidad del tanque.

Además se necesita modificar el programa, de manera que se puedan ingresar n cantidad de motos, ya que se necesita que podamos brindar la siguiente información:
•Cantidad de motos con menos de 1000km
•Cantidad de motos patentadas en 2021.

Le dejamos la solución del ejercicio dado, pero se les recomienda intentar varias veces hasta que salga la solución antes de ver la misma.

Link a la resolución del ejercicio propuesto:

https://drive.google.com/file/d/1kNCmj0-qCiNJKhyBZQEYQQHN8oO8moq7/view?usp=sharing

El ejercicio está preparado para una moto, ¿pueden modificarlo para recibir un listado de motos?

5. Abstracción

Es la capacidad para extraer las propiedades esenciales de un concepto e identificar comportamientos comunes para definir nuevos tipos de entidades en el mundo real.

Abstracción funcional:

Es lo que sabemos que un objeto puede hacer, por ejemplo en el caso de un auto , acelerar, frenar, doblar.

Abstracción de datos:

Son los atributos que tiene un determinado tipo de objetos, por ejemplo en el caso de un auto, marca, modelo, patente, color.

6. Herencia

Es la propiedad que permite definir una clase a partir de otra clase relacionado que suponga alguna de las siguientes relaciones:

Especialización:

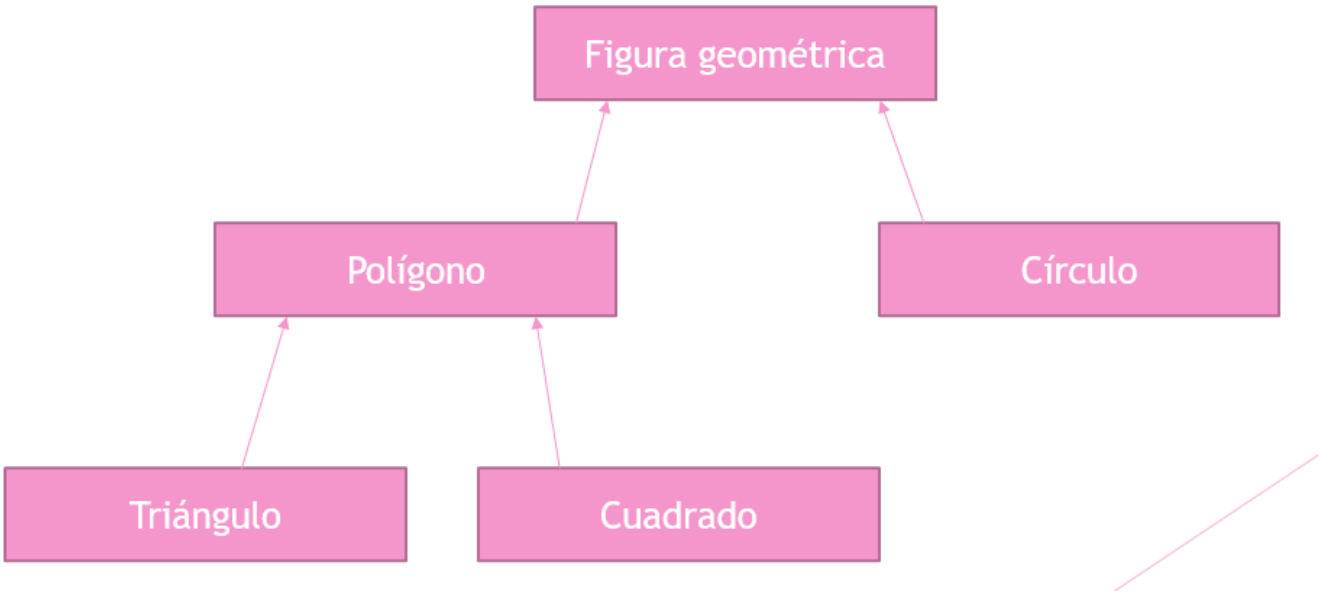
La clase Auto como especialización de Vehículo.

Generalización:

La clase Vehículo como generalización de la clase Auto.

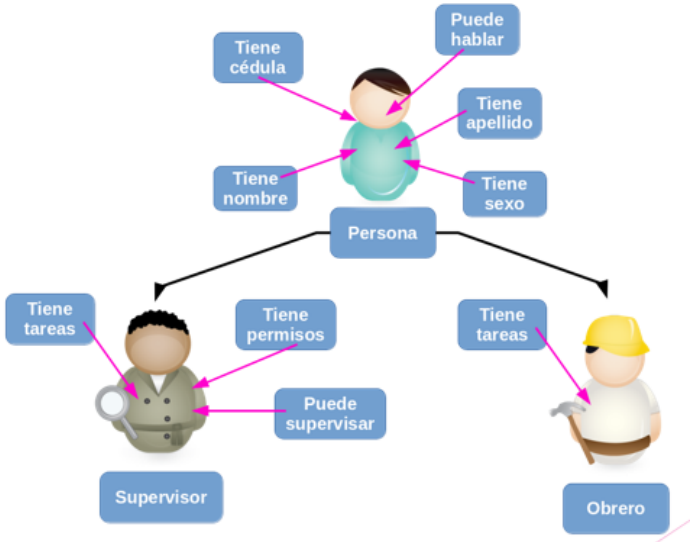
Es un mecanismo para la reutilización de código.
Ejemplo:
Si una clase B (celular) hereda de una clase A (productos) implica que:
B incorpora la estructura (atributos) y comportamientos (métodos)de la clase A.
B puede añadir nuevos atributos.
B puede añadir nuevos métodos.
B puede redefinir métodos (sobre escritura u overwriting).

La herencia organiza las clases en una estructura jerárquica formando jerarquía de clases.
Ejemplo:



Python permite la herencia múltiple, es decir, se puede heredar de múltiples clases. La herencia múltiple es la capacidad de una subclase de heredar de múltiples súper clases.

Lo vemos más gráfico en un ejemplo:



6.1. Ejercicio práctico de Herencia

Crear la clase Triciclomotor que herede de la clase Motovehículo y le añada el atributo de capacidad de carga y características especiales.

Le dejamos la solución del ejercicio dado, pero se les recomienda intentar varias veces hasta que salga la solución antes de ver la misma.

Link a la resolución del ejercicio propuesto:

https://drive.google.com/file/d/12pRLGaXl3r6Jmbc36SLO-h0iW6ldA64V/view?usp=sharing

7. Encapsulamiento y Ocultamiento de la información

Encapsulamiento:

La encapsulación es una forma de darle uso exclusivo a los comportamientos o atributos que posee una clase, es decir, protege esos atributos y comportamientos para que no sean usados de manera externa.

En python para hacer que un atributo o comportamiento sea privado tenemos que colocar un par de guiones bajos antes del nombre del atributo o comportamiento "__nombre".

Para empezar nuestro ejemplo de encapsulación vamos a crear una clase que llamaremos "Ejemplo" y dentro de ella declaramos un método al que llamaremos "publico" que contendrá un **return** que solo mostrara una cadena de texto que dirá "Soy un método público a la vista de todo":

```
class Ejemplo():

    def publico(self):
        return "Soy un método público, a la vista de todo"
```

Ahora declaramos un método que se llame "privado" pero antes de su nombre pondremos un par de guiones bajos "__" y dentro del método una cadena de texto como en el método anterior:

```
class Ejemplo():

    def publico(self):
        return "Soy un método público, a la vista de todo"
    def __privado(self):
        return "Soy un metodo privado, para ti no existo"
```

Ya con todo esto creamos un objeto perteneciente a la clase ejemplo y procedemos a imprimir los dos métodos hemos creado:

```
class Ejemplo():

    def publico(self):
        return "Soy un método público, a la vista de todo"
    def __privado(self):
        return "Soy un metodo privado, para ti no existo"

objeto = Ejemplo()

print(objeto.publico())

print(objeto.__privado())
```

Si ejecutamos el ejemplo nos mostrara algo parecido:

```
Soy un método público, a la vista de todo
Traceback (most recent can last):
 File "encapsulación.py", line 12, in (module)
print(objeto.__private())
AttributeError: 'Ejemplo' object has no attribute '__privado'
```

Como puedes ver al ejecutarlo si nos muestra el contenido del método publico pero a la hora de mostrar el método privado nos dice que tal método no existe pero en realidad si solo que por ser privado no puede ser mostrado externamente.

Ocultamiento:

El ocultamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas. Esto asegura que otros objetos no pueden cambiar el estado interno de un objeto de maneras inesperadas.

Cada tipo de objeto expone una interfaz a otros objetos que especifica cómo pueden interactuar con los objetos de una clase.

Es un concepto relacionado con la seguridad de los datos.

8. Polimorfismo

El polimorfismo nos permite programar de manera general en lugar de programar de manera específica.

Hay cuatro técnicas, cada una de las cuales permite una forma distinta de reutilización de código, que facilita a su vez el desarrollo rápido, la confianza, facilidad de uso y mantenimiento del software:
•Sobrecarga
•Sobreescritura
•Variables polimórficas
•Genericidad

Sobrecarga (overloading, polimorfismo ad-hoc)

•Un solo nombre de método y muchas implementaciones distintas.
•Las funciones sobrecargadas normalmente se distinguen en tiempo de compilación por tener distintos parámetros de entrada y/o salida.

Sobreescritura (overriding, polimorfismo de inclusión):

•Tipo especial de sobrecarga que ocurre dentro de relaciones de herencia en métodos con enlace dinámico.
•Dichos métodos, definidos en clases base, son refinados o reemplazados en las clases derivadas.

Variables polimórficas (polimorfismo de asignación):

Variable que se declara como de un tipo pero que referencia en realidad un valor de un tipo distinto (normalmente relacionado mediante herencia).

Genericidad (plantillas o templates):

•Clases o métodos parametrizados (algunos tipos se dejan sin definir).
•Forma de crear herramientas de propósito general (clases, métodos) y especializarlas para situaciones específicas.

Ejemplos:

Variables polimórficas:

•Cuenta pc=new CuentaJoven();

Genericidad:

•Lista<Cliente>
•Lista<Articulo>
•Lista<Alumno>
•Lista<Habitacion>

Sobrecarga:

•Factura: imprimir()
•Factura: imprimir(int numCopias)
•ListaCompra: imprimir()

Sobreescritura:

•Cuenta: abonarIntereses()
•CuentaJoven: abonarIntereses()

8.1. Ejercicio práctico de Polimorfismo

En este simple, pero claro ejemplo, veremos como el polimorfismo puede brindarnos la posibilidad de tomar más de una forma una función:

Crear la función "Desplazamiento" y utilizarla para desplazarse en tres tipos de vehículos diferentes.

Link de resolución:

https://drive.google.com/file/d/1-gMTao0Vu9PwTa24JPJYvLQlIgIVTZCC/view?usp=sharing