Unidad 2
Paradigma orientado a objetos, clases y objetos

Prof: Lic. Rosales Pablo (prosales@unpata.edu.ar)
Materia: Programación Orientada a Objetos
Año: 2024

Paradigma Orientado a Objetos

"*Enviar mensajes entre objetos para simular una evolución temporal de un conjunto de fenómenos de la vida real*"

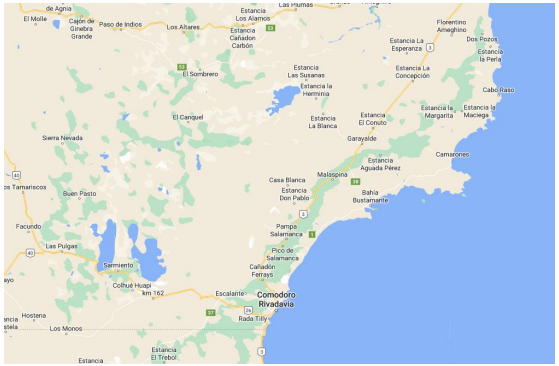
El éxito del paradigma es la importancia de los conceptos. Un programa orientado a objetos se construye desde el principio en conceptos, que son importantes en el dominio del problema de interés. De esa forma, deja todos los tecnicismos necesarios de la programación en segundo plano.

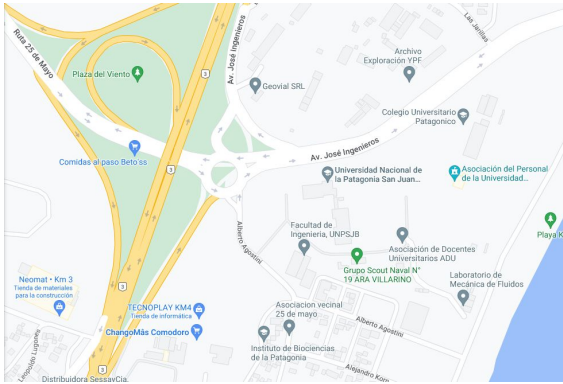
¿Qué “*conceptos*”?

Cuatro pilares fundamentales del paradigma orientado a objetos
Abstracción, Encapsulamiento, Herencia y Polimorfismo

Abstracción

Información nueva aparece y es más específica, calles locales etc...





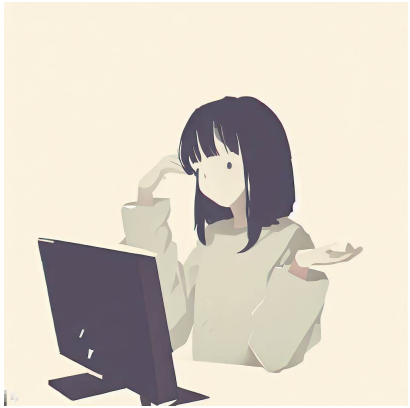
La información específica se abstrae, se omite, no hay espacio.

Abstracción

Fundamentalmente, una persona promedio sólo utiliza herramientas o componentes simples para crear, entender y administrar sistemas complejos.

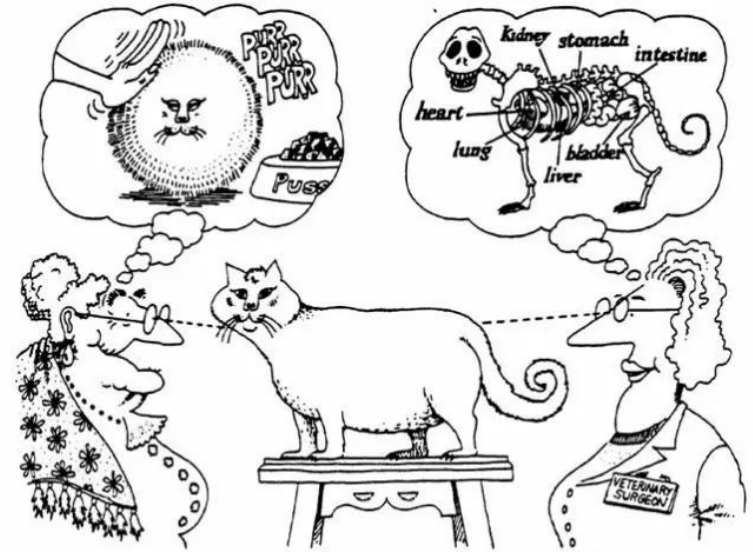
- Usar una computadora.
- Manejar un auto.
- Mantener comida en una heladera o calentar en microondas.

Por medio de la abstracción creamos modelos de los sistemas reales.



Abstracción

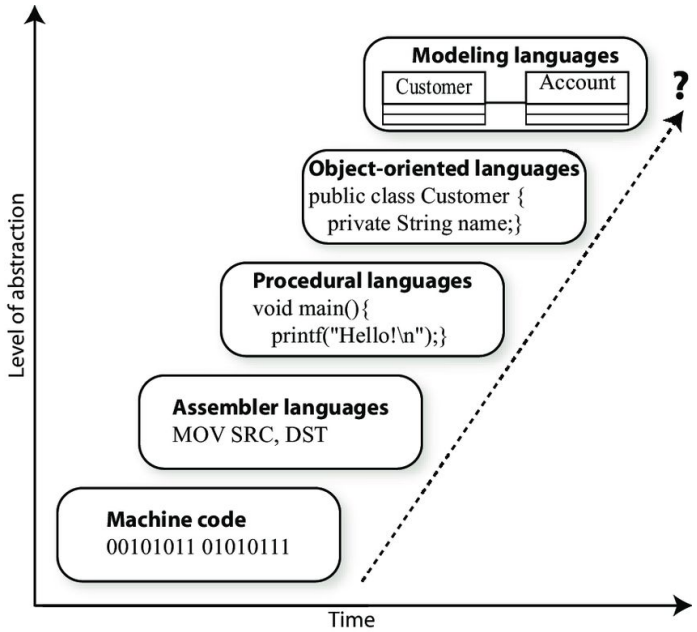
Es la supresión u ocultación deliberada de algunos detalles de un proceso o artefacto, con el fin de resaltar más claramente otros aspectos, detalles o estructuras.



Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.

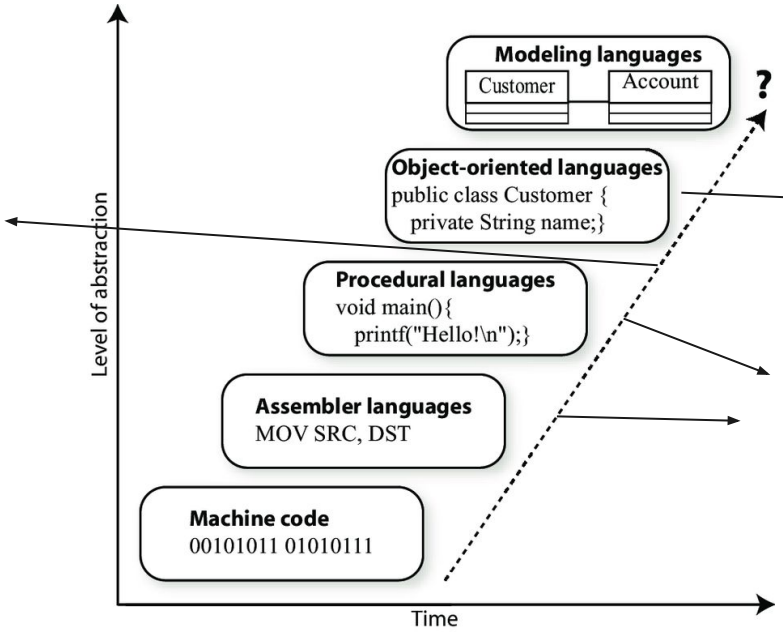
¿Por qué usar abstracciones?

- Más fácil de pensar (se esconde lo que no importa)
- Se protege lo que no debería ser visto o modificado.
- La división conceptual permite reemplazar los componentes fácilmente.



Reichwein, Axel. (2012). Application-specific UML profiles for multidisciplinary product data integration.

Centrado en datos:
los módulos y TAD evolucionan a un punto de vista más enfocado al dato.



Centrado en servicios:
que servicio el objeto puede ofrecer al resto del programa.

Centrado en funciones:
programador se enfoca en cómo la tarea debe ser cumplida.

Clases

es un “molde” para crear objetos (instancias) que comparten atributos (datos) y comportamiento (métodos).

```python
class CuentaCorriente:
    def __init__(self, nombre, saldo=0):
        self.nombre = nombre
        self.saldo = saldo

    def depositar(self, monto):
        if monto > 0:
            self.saldo += monto
            print(f"Depositado ${monto}. Nuevo saldo: ${self.saldo}")
        else:
            print("Monto a depositar invalido.")

    def retirar(self, monto):
        if 0 < monto <= self.saldo:
            self.saldo -= monto
            print(f"Retira ${monto}. Nuevo saldo: ${self.saldo}")
        else:
            print("Saldo insuficiente o monto invalido.")

    def get_saldo(self):
        return self.saldo


cuenta = CuentaCorriente("Juana", 1000)

print(f"Dueño de cuenta: {cuenta.nombre}")
print(f"Saldo inicial: ${cuenta.get_saldo()}")

cuenta.depositar(500)
cuenta.retirar(200)
cuenta.retirar(1500)

print(f"Saldo final: ${cuenta.get_saldo()}")
```

Las clases deben ser:

Altamente cohesivas: La clase representa una única entidad (no varias), los datos y lo que puede hacer con ellos están muy relacionados entre sí.

Mínimamente acoplada: La clase limita sus interacciones con otras, se limita a relacionarse con aquellas que son necesarias y para las cuales fue diseñada.

Encapsuladas: mantiene resguardada su información y como la manipula de las otras clases y permite el acceso de manera intencionada.

Estas son características de diseño a aspirar, se les van a presentar situaciones en la que es difícil de aplicar.

Encapsulamiento

Consiste en combinar datos y comportamiento en un paquete y ocultar los detalles de la implementación del usuario del objeto.

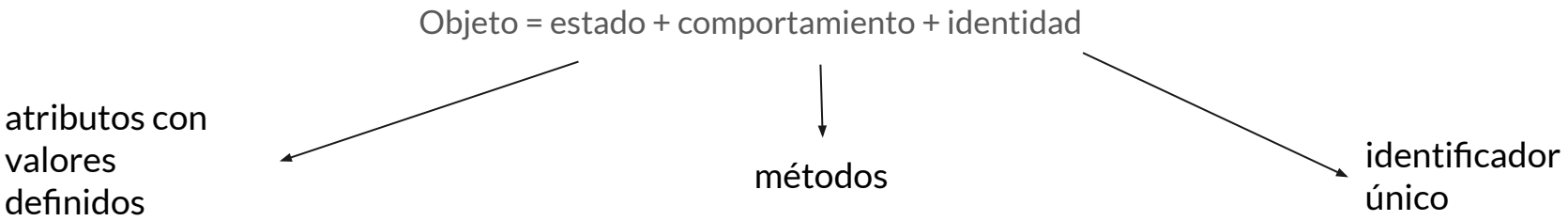
Oculta lo que hace un objeto de lo que hacen otros objetos “del mundo exterior”, también llamado ocultación de datos.

Una clase se puede ver como una caja negra.

¿Clase u Objeto?

La **Clase** es un modelo de una entidad definida por su estado y comportamiento. No es un ejemplo específico de una entidad, es una “plantilla” para todas las instancias de esta entidad. Solo hay una definición de una clase.

Un **Objeto** es “encarnación” o instancia de una clase. Una ocurrencia concreta de esta clase. Pueden existir varias instancias de la clase al mismo tiempo.



Modificadores de acceso o visibilidad (python)

Public: atributos o métodos que no tienen de prefijo un guión bajo (_), se considera que pueden ser accedidos desde cualquier lado, adentro y afuera de la clase.

Protected: atributos o métodos que tiene un guión bajo (_) como prefijo, se deben usar en la propia clase y subclases. (pj: _attr_protected)

Private: atributo con doble guión bajo (__) indica que no debería usarse desde afuera de la clase.

python no limita esto, la visibilidad está planteada como una convención de nombres.

```python
class MiClase:
    def __init__(self):
        self.public_attr = "attr publico"
        self._protected_attr = "attr protegido"
        self.__private_attr = "attr privado"

    def public_metodo(self):
        print("Este es un metodo publico.")

    def _protected_metodo(self):
        print("Este es un atributo protegido.")

    def __private_metodo(self):
        print("Este es un metodo privado.")

obj = MiClase()

print(obj.public_attr)         # bien
print(obj._protected_attr)     # no recomendado
#print(obj.__private_attr)     # Error AttributeError
print(obj._MiClase__private_attr) # mangled name, no usar!

obj.public_metodo()           # bien
obj._protected_metodo()       # no recomendado
# obj.__private_metodo()      # Error AttributeError
print(obj._MiClase__private_metodo()) # mangled name, no usar!
```

Modificadores de acceso o visibilidad (java)

Public: desde cualquier parte del programa.

Protected: desde el mismo paquete y por subclases (aun fuera del paquete).

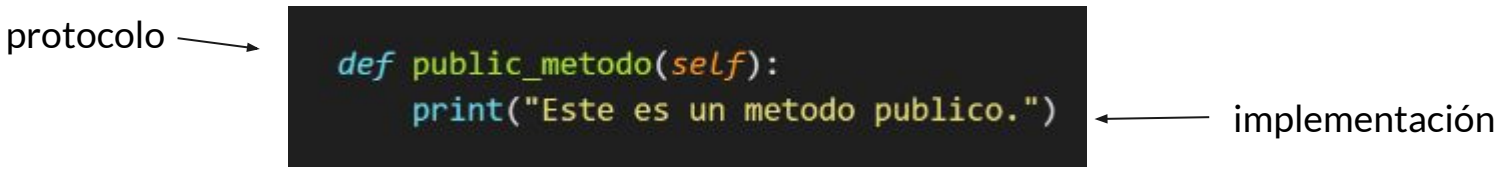
Default: desde el mismo paquete (no desde subclase en diferente paquete).

Private: solo desde la misma clase.

Mensajes y métodos

Los mensajes definen el **protocolo de comunicación** e indica que sabe hacer el objeto.

Los métodos son la implementación del mensaje (**como** se hace).



Particularidades de métodos en Python

self: primer parámetro en métodos, permite acceder a los valores de instancia.

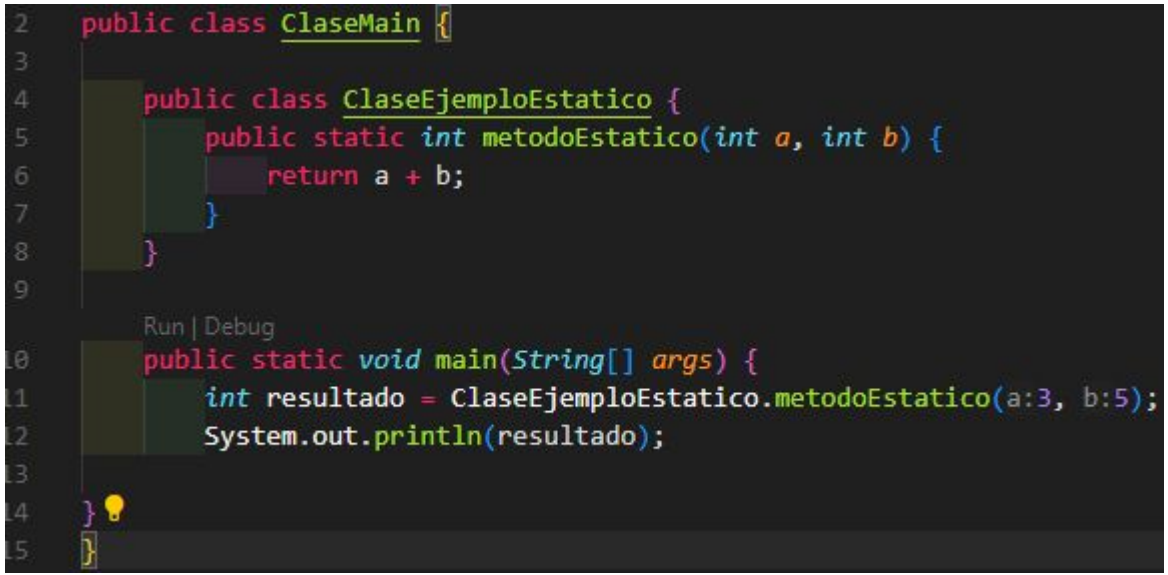
no hace falta instancia

cls: primer parámetro en un método de clase, los @classmethod (decorator) o métodos de clase realizan acciones que no involucran los valores de un objeto.

@staticmethod: sin parámetro cls o self, no tiene acceso a atributos de clase ni instancia. Para funciones de "utilidad".

```python
class MiClase:
    attr_clase = "Un atributo de clase"

    def __init__(self, valor):
        self.attr_instancia = valor

    def metodo_instancia(self):
        print("Este es un metodo de instancia.")
        print("atributo de instancia:", self.attr_instancia)

    @staticmethod
    def metodo_estatico():
        print("Este es un metodo estatico.")

    @classmethod
    def metodo_clase(cls):
        print("Este es un metodo de clase.")
        print("Atributo de clase:", cls.attr_clase)

obj = MiClase("Holaa!")

obj.metodo_instancia()
MiClase.metodo_estatico()
obj.metodo_clase()
MiClase.metodo_clase()
```

método estático en Java



¿pass-by-value o pass-by-reference? no, pass-by-object

para objetos **inmutables**, al pasarlo como parámetro se crea una nueva referencia en el alcance de la función, pero cualquier modificación resulta en la creación de un nuevo objeto, no pudiendo modificar el original.

algunos ejemplos de objetos inmutables:

int, float, complex, str, bytes, bool, Tuples, Frozen Sets,

Namedtupled, Bytes, None,

```python
def modifica_lista(lst):
    lst.append(4)

def modifica_string(s):
    s += " mundo"

mi_lista = [1, 2, 3]
modifica_lista(mi_lista)
print(mi_lista)  # Output: [1, 2, 3, 4]

mi_string = "Hola"
modifica_string(mi_string)
print(mi_string)  # Output: Hola (no "Hola mundo")
```

¿pass-by-value o pass-by-reference? no, pass-by-object

Para objetos **mutables** se pasa una referencia, por lo tanto el objeto original es modificado, también cambia.

Por ejemplo:

Lists, Dictionaries, Sets, Clases definidas por usuarios.

```python
def modifica_lista(lst):
    lst.append(4)

def modifica_string(s):
    s += " mundo"

mi_lista = [1, 2, 3]
modifica_lista(mi_lista)
print(mi_lista)  # Output: [1, 2, 3, 4]

mi_string = "Hola"
modifica_string(mi_string)
print(mi_string)  # Output: Hola (no "Hola mundo")
```

Constructores

Se invoca automáticamente luego de que la instancia es creada, es responsable de setear el estado inicial del objeto.

siempre se llama __init__(self) y tiene como primer parámetro a self.

Existe un método __new__() que se llama antes que el objeto se construya, por lo tanto no tiene un parámetro self. No es normalmente usado.

En Python no existe la sobrecarga del constructor.

Destructores

Existe la función __del__(self) que es invocada cuando se está por eliminar el objeto.

Se puede usar para operaciones de limpieza como cerrar archivos y liberar recursos.

Cuando ya no hay referencias al objeto o se borra explícitamente con la palabra reservada del, el metodo __del__ se ejecuta.

```python
class MiClase:
    def __init__(self):
        print("Constructor")

    def __del__(self):
        print("Destructor")

obj = MiClase()
del obj
```

Getters y Setters

Con el decorator @property podemos definir un método para que se comporte como un get.

Con el @<identificador>.setter, en este caso @mi_attr.setter definimos el set del atributo.

salida por consola:

```
Set de mi attr
Get de mi attr
Valor del atributo: 77
```

```python
class MiClase:
    def __init__(self):
        self.__mi_attr = None  # privado con valor inicial None

    @property
    def mi_attr(self):
        print("Get de mi attr")
        return self._mi_attr

    @mi_attr.setter
    def mi_attr(self, valor):
        print("Set de mi attr")
        self._mi_attr = valor

obj = MiClase()

# uso el atributo
obj.mi_attr = 77    # Get de mi attr
valor = obj.mi_attr # Set de mi attr

print("Valor del atributo:", valor)  # Valor del atributo: 77
```

A implementar:

Crear un modelo agregando clases: por ejemplo

persona y banco.

Hacer diagrama UML (ver cheat-sheet)

Codificar teniendo en cuenta que cada clase va un archivo y en lo posible, implementar paquetes.

```python
class CuentaCorriente:
    def __init__(self, nombre, saldo=0):
        self.nombre = nombre
        self.saldo = saldo

    def depositar(self, monto):
        if monto > 0:
            self.saldo += monto
            print(f"Depositado ${monto}. Nuevo saldo: ${self.saldo}")
        else:
            print("Monto a depositar invalido.")

    def retirar(self, monto):
        if 0 < monto <= self.saldo:
            self.saldo -= monto
            print(f"Retira ${monto}. Nuevo saldo: ${self.saldo}")
        else:
            print("Saldo insuficiente o monto invalido.")

    def get_saldo(self):
        return self.saldo


cuenta = CuentaCorriente("Juana", 1000)

print(f"Dueño de cuenta: {cuenta.nombre}")
print(f"Saldo inicial: ${cuenta.get_saldo()}")

cuenta.depositar(500)
cuenta.retirar(200)
cuenta.retirar(1500)

print(f"Saldo final: ${cuenta.get_saldo()}")
```

Style Guide Python

Indentación: usar 4 espacios por nivel.

Longitud máxima por línea: 79 caracteres para códigos y comentarios.

Importaciones: en líneas separadas agrupadas según (Standard library, Related third-party, specific local app/library). no usar wildcard *.

Espacios en blaco: evitar al inicio y al fin de lineas, evitar alienar asignaciones, solo un espacio entre asignaciones.

Convención de nombres: minúscula con guión bajo para archivos, funciones, métodos , variables y módulos (snake_case). CapWords para clases (CamelCase) y guión bajo inicial para visibilidad _protected, __private.

ver más en: https://peps.python.org/pep-0008/

Extensiones utiles VSCode

UMLLet: para hacer diagramas UML (crear archivo .uxf)

Polyglot: para dar soporte a los archivos .ipynb tipo Jupyter Notebooks.