

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación



IIC2115 - Programación como Herramienta para la Ingeniería

Fundamentos: POO y EDD

Profesor: Hans Löbel

Agenda para hoy

- Programación Orientada a Objetos (POO): cómo modelar un problema y organizar un programa a través de elementos que se comunican.
- Estructuras de Datos: objetos que manejan datos de manera más eficiente y/o efectiva para determinadas situaciones/problemas.
- Resolución de problemas y uso de IA en este capítulo

Cuando hablamos de un curso, ¿en qué estamos pensando?

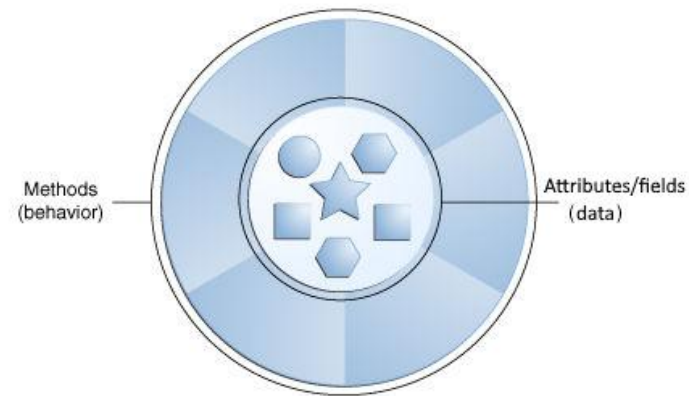


- Todas estas maneras de “modelarlo” representan distintas abstracciones del concepto curso, cada una más o menos adecuada para distintas tareas.
- En cualquiera de estos casos, “un curso” siempre tendrá elementos que lo constituyen o definen (partes, atributos), y formas en que el curso cambia o evoluciona

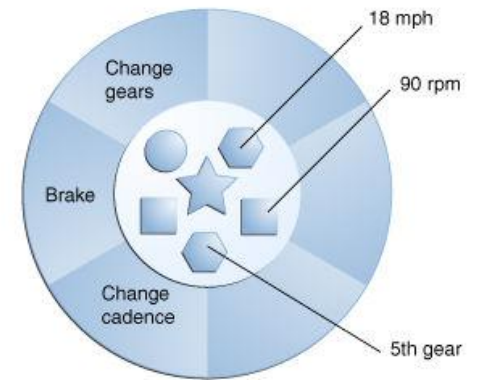
Objetos de software capturan estas ideas

En el desarrollo de software, los objetos modelan **entidades**, **abstracciones** o cualquier elemento relevante de un programa. Se estructuran como una colección de **datos** que además tiene asociado **comportamientos**.

- Datos: **describen** el estado y/o composición de los objetos. Se les conoce como **atributos** o **campos** del objeto.
- Comportamientos: **representan acciones** que realiza el objeto, o realizan sobre él, que pueden generar cambios en su estado. Se les conoce como **métodos** del objeto.



(a) A software object



(b) Bicycle modelled as a software object

Ejemplo: datos y comportamiento

Clase: Auto	
Datos	Comportamiento
Marca	Viajar
Modelo	Cargar el estanque
Color	Realizar mantención
Año	
Motor	
Kilometraje	
Ubicación actual	

¿Qué es entonces OOP?

La programación orientada a objetos es una forma de programar, que consiste en representar los elementos de un problema como “objetos” que tienen datos y comportamientos, y que deben relacionarse entre sí para construir la solución buscada.

Para definir un objeto, creamos una plantilla llamada **clase**

Cada objeto es una **instancia** de la clase Auto

Objeto 1



Objeto 2



Objeto 3



Clase **Auto**

```
1 class Departamento:
2     def __init__(self, _id, mts2, valor, num_dorms, num_banos):
3         self._id = _id
4         self.mts2 = mts2
5         self.valor = valor
6         self.num_dorms = num_dorms
7         self.num_banos = num_banos
8         self.vendido = False
9
10    def vender(self):
11        if not self.vendido:
12            self.vendido = True
13        else:
14            print("Departamento {} ya se vendió".format(self._id))
```



```
1 d1 = Departamento(_id=1, mts2=100, valor=5000, num_dorms=3, num_banos=2)
2 print(d1.vendido)
3 d1.vender()
4 print(d1.vendido)
5 d1.vender()
```

False

True

Departamento 1 ya se vendió

```
1 d2 = Departamento(_id=2, mts2=185, valor=4000, num_dorms=2, num_banos=1)
2 d3 = Departamento(_id=1, mts2=100, valor=5000, num_dorms=3, num_banos=2)
3 d3.vender()
4 d4 = d1
5
6 print(d1 == d2)
7 print(d1 == d3)
8 print(d1 == d4)
9
10 d4.vendido = False
11 print(d1.vendido == d4.vendido)
```

```
1 d2 = Departamento(_id=2, mts2=185, valor=4000, num_dorms=2, num_banos=1)
2 d3 = Departamento(_id=1, mts2=100, valor=5000, num_dorms=3, num_banos=2)
3 d3.vender()
4 d4 = d1
5
6 print(d1 == d2)
7 print(d1 == d3)
8 print(d1 == d4)
9
10 d4.vendido = False
11 print(d1.vendido == d4.vendido)
```

False

False

True

True

Un concepto fundamental es el de **interfaz** de un objeto

Existen atributos de los objetos que no necesitan ser visualizados ni accedidos por los otros objetos con que se interactúa.



Un concepto fundamental es el de **interfaz** de un objeto

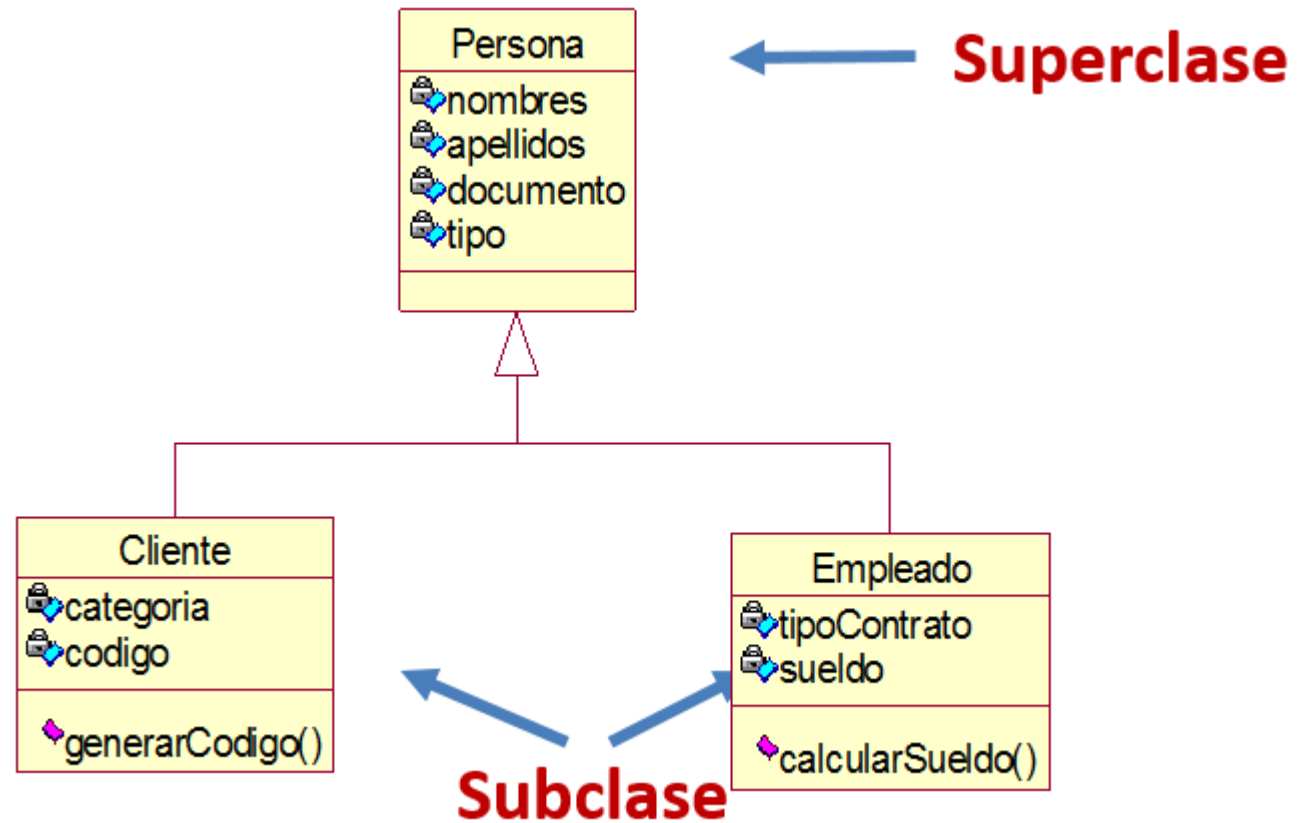
Lo mismo ocurren con los métodos, no todos tienen que ser accesibles para cualquier otro objeto.



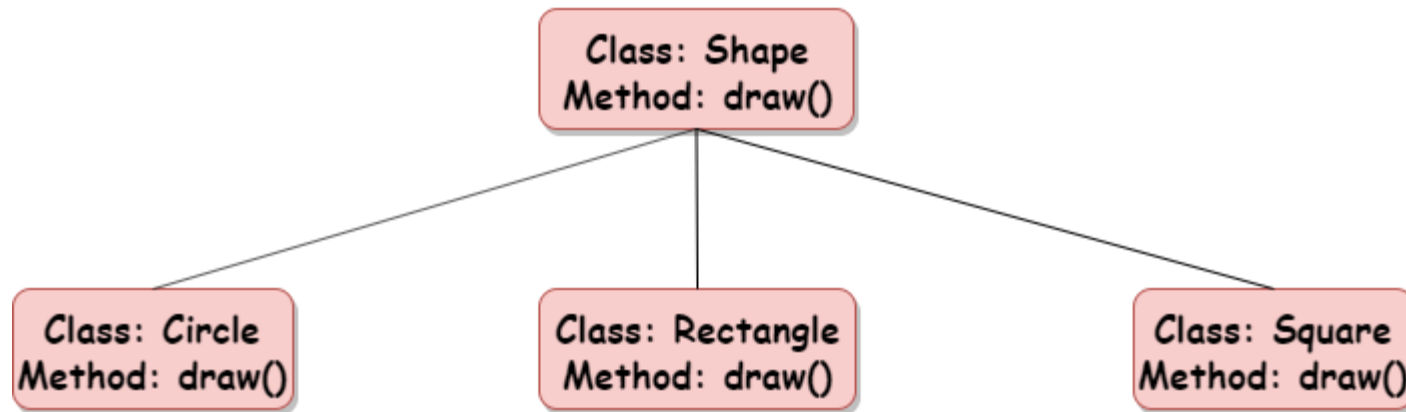
Interface
Turn on Turn off Volume up Volume down Switch to next channel Switch to previous channel
Current channel Volume level

```
1 class Televisor:
2     ''' Clase que modela un televisor.
3     '''
4
5     def __init__(self, pulgadas, marca):
6         self.pulgadas = pulgadas
7         self.marca = marca
8         self.encendido = False
9         self.canal_actual = 0
10
11     def encender(self):
12         self.encendido = True
13
14     def apagar(self):
15         self.encendido = False
16
17     def cambiar_canal(self, nuevo_canal):
18         self._codificar_imagen()
19         self.canal_actual = nuevo_canal
20
21     def __codificar_imagen(self):
22         print("Estoy convirtiendo una señal eléctrica en la imagen que estás viendo.")
```

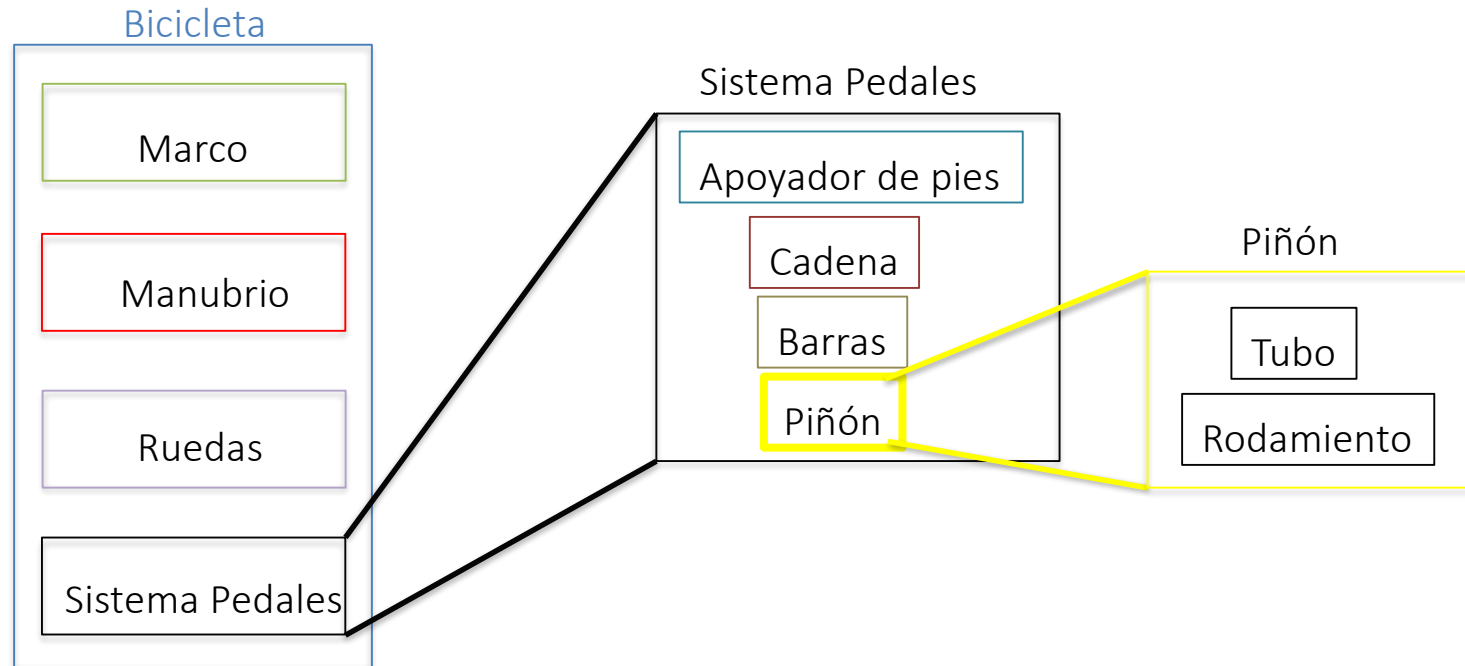
Herencia nos permite modelar **clases** similares sin reescribir todo de nuevo, facilitando la especialización



Polimorfismo es la capacidad que tienen las clases de cambiar la implementación de los métodos previamente definidos



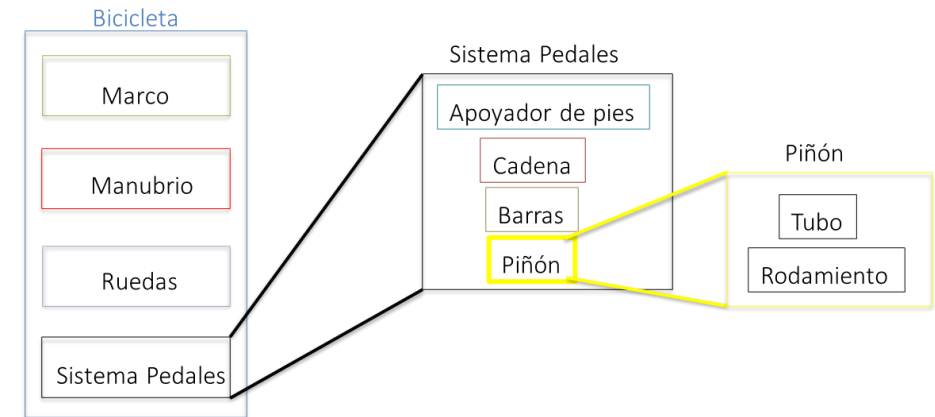
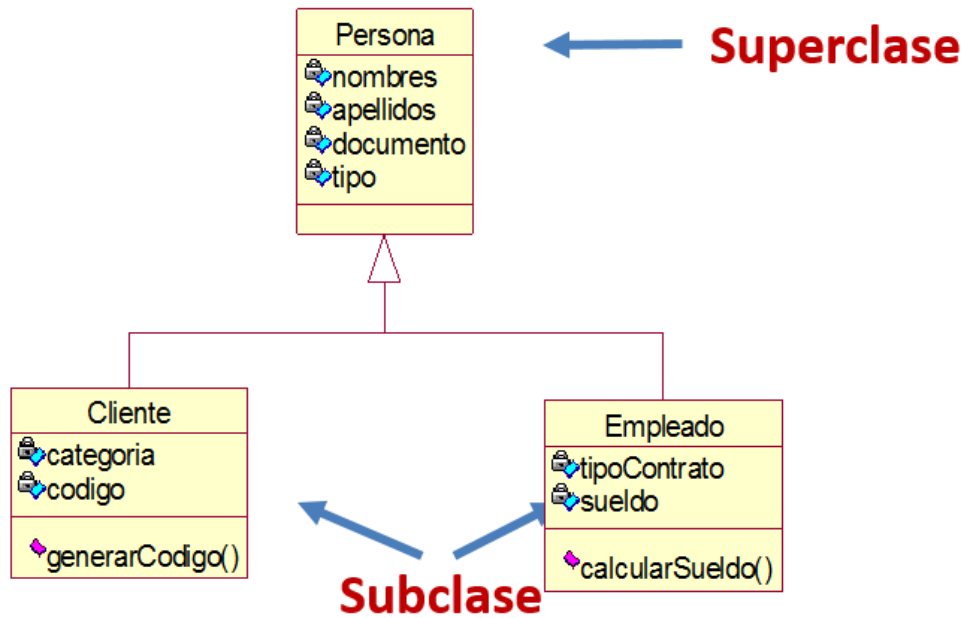
Es posible modelar **objetos** como atributos de otros objetos, mediante **agregación o composición**



Agregación: atributo existe de manera independiente al contenedor

Composición: atributo no puede existir de independiente del contenedor

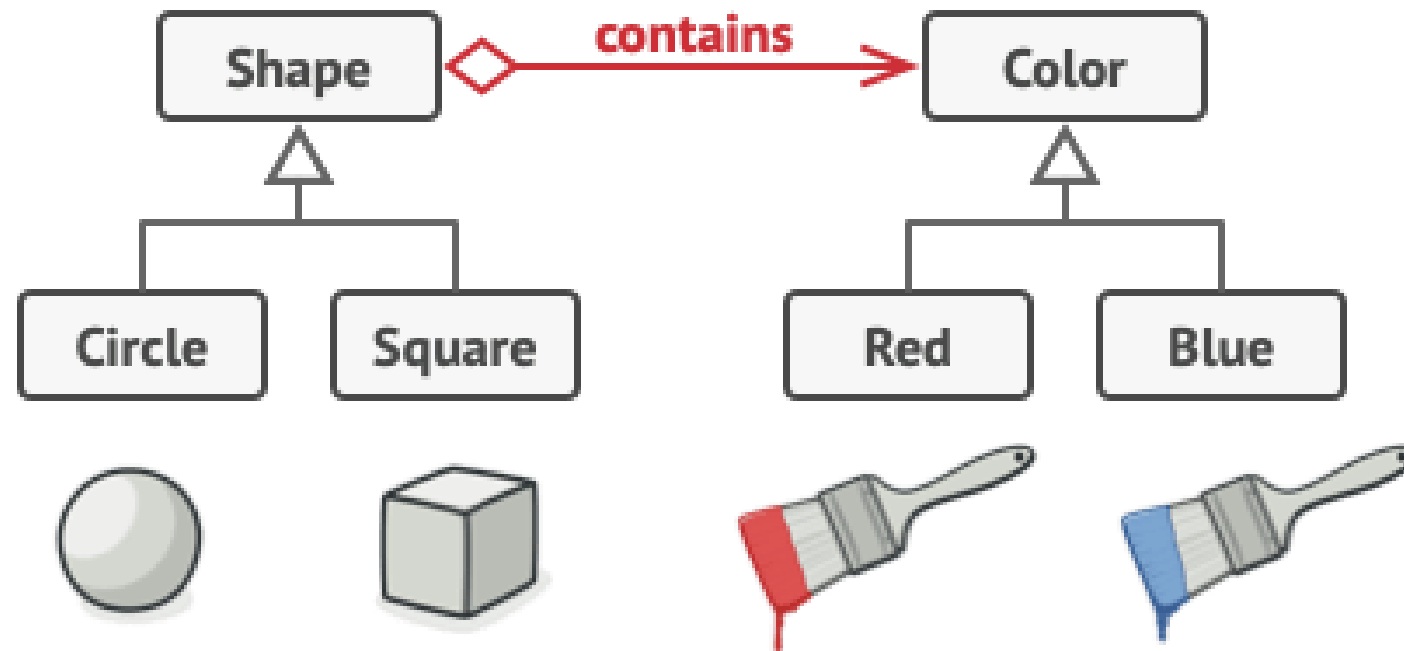
¿Cómo se comparan herencia y agregación/composición?



¿Cómo se comparan herencia y agregación/composición?

- NO TIENEN MUCHO QUE VER EN REALIDAD!
- Si bien ambos son mecanismo para modelar, estructuralmente difieren de manera fundamental.
- **Herencia** busca facilitar la **especialización** de las clases, sin requerir repetir código.
- **Agregación y composición** buscan aumentar el nivel de **abstracción** de las clases, al permitir tipos de dato complejos (otras clases) como atributos.
- Ambos son fundamentales y se usan en conjunto comúnmente.

¿Cómo se comparan herencia y agregación/composición?



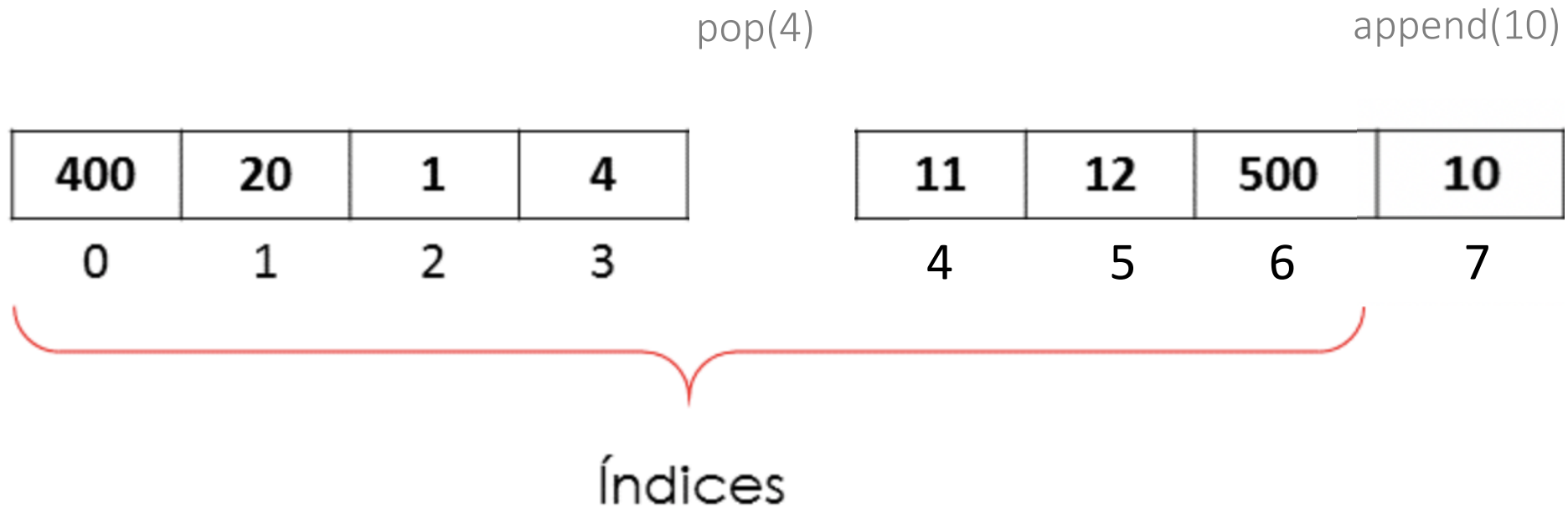
Movámonos ahora a las estructuras de datos (EDD)

Son **tipos de dato especializados**, diseñados para **agrupar, almacenar o acceder** a la información de manera más **eficiente** que un tipo de dato básico (como int, float, etc). Algunos ejemplos son los siguientes:

- Clases
- Listas
- Tuplas
- Diccionarios
- Árboles

Listas


- Las listas son estructuras que guardan datos de forma **ordenada**.
- Son mutables (modificables).



Tuplas

- Similares a las listas, permiten manejar datos de forma ordenada.
- Al igual que las listas, se accede a los datos mediante índices basados en el orden que fueron ingresados.
- A diferencia de las listas, son **inmutables**.

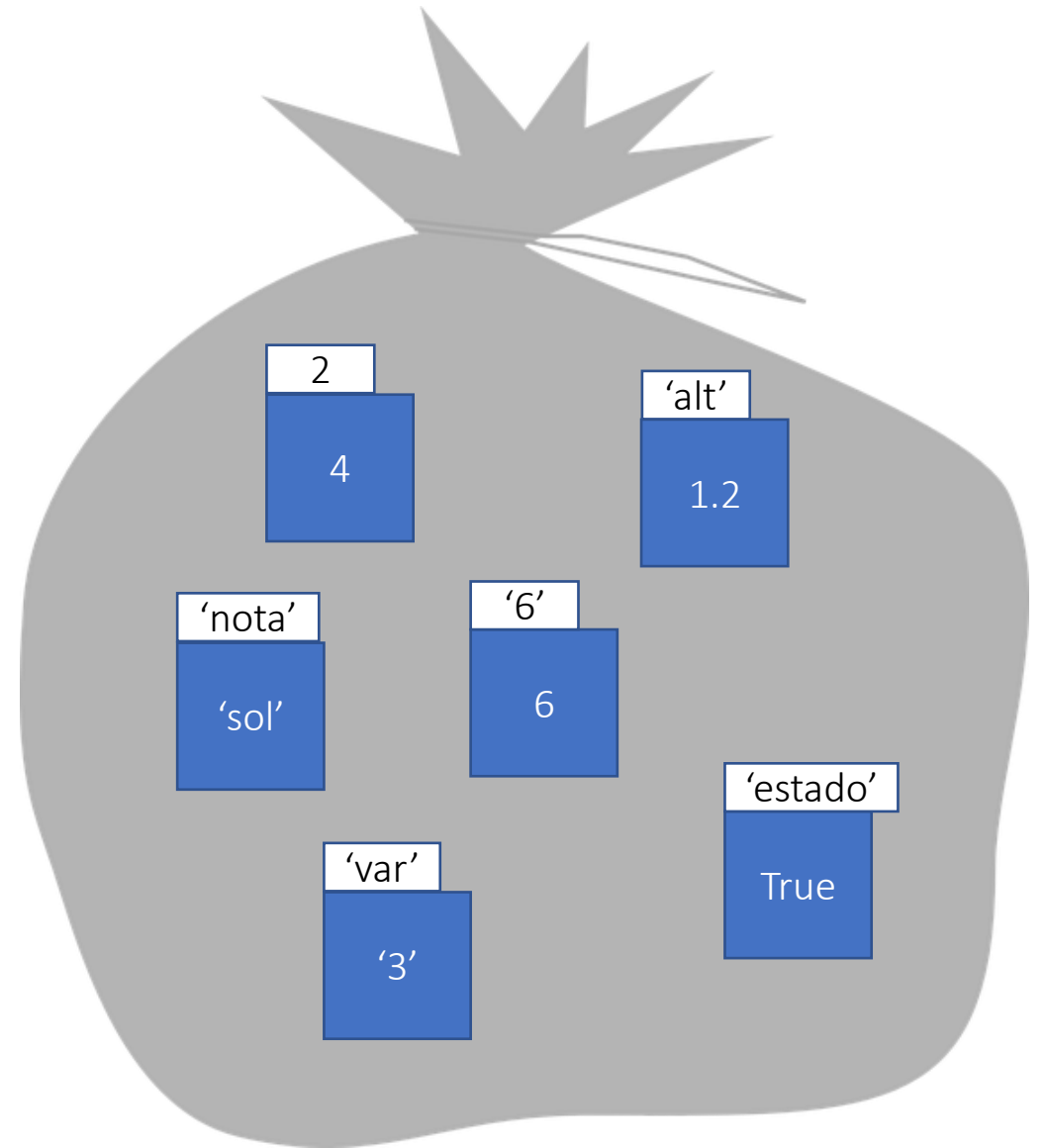
400	20	1	4	10	11	12	500
0	1	2	3	4	5	6	7



Índices

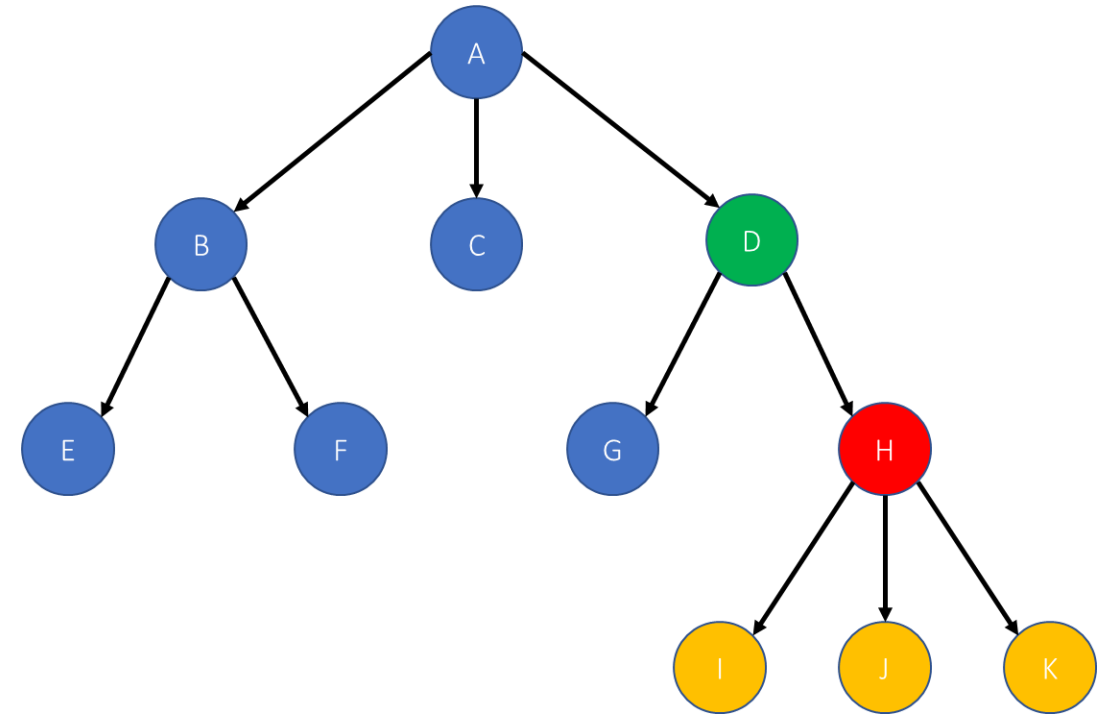
Diccionarios

- Permiten almacenar datos basados en una asociación de pares de elementos, a través de una relación **llave-valor**.
- Acceso a valores a través de la llave es instantáneo, no se necesita realizar una búsqueda (análogo a un índice).
- Se prefiere a una lista cuando el caso de uso más común no implica revisar todos los elementos, sino solo algunos fácilmente encontrables a través de la llave.



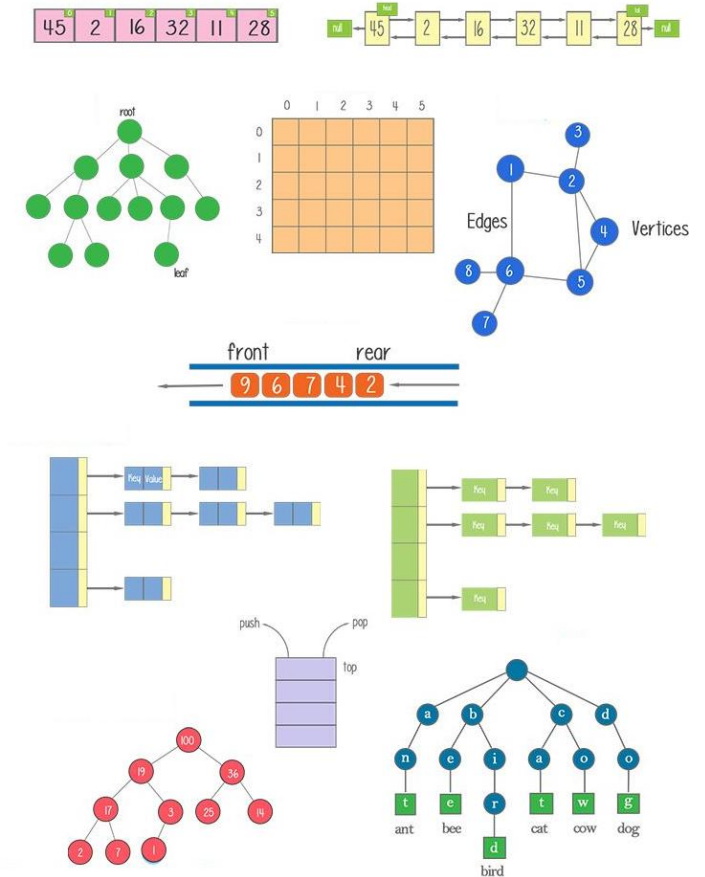
Árboles

- Son útiles cuando los datos tienen una estructura jerárquica: rutas en una red de transporte o un organigrama.
- Para buscar cosas rápido: autocompletar de búsqueda en Google
- Para encontrar el camino más corto en una red.
- Para tomar decisiones paso a paso: en IA, por ejemplo, cuando se analizan varias opciones antes de elegir la mejor.
- Mucho de esto se logra mediante búsqueda eficiente en árboles.



Un breve y somero resumen de EDD

- Las estructuras de datos (incluyendo las clases) corresponden a tipos de dato especializados, **diseñados para organizar, almacenar y/o acceder la información de manera más eficiente** que un tipo de dato básico.
- La elección adecuada de la estructura de datos, o la manera de organizar las clases, es fundamental para el desarrollo de un buen programa y muchas veces es la única posibilidad para solucionar un problema de forma realista.
- Pero siempre es conveniente pensar primero en una solución básica a los problemas, y luego incorporar las estructuras donde corresponda.



Cerremos la clase con la forma de resolver los problemas para este capítulo del curso

- División y estructuración de los problemas
- Especificación de *prompts*
- Testeo del código
- Sobre la IA: no usar una cuenta compartida, parece una buena idea, pero siempre falla

Usemos el siguiente problema como ejemplo guía

Escriba un programa que permita ejecutar una serie de procesos docentes de fin e inicio de semestre para una institución de educación superior.

A modo general, la división siempre dependerá de la temática del capítulo

- División progresiva del problema en subproblemas más simples: **dividir y conquistar**
- Dada la temática del capítulo, el foco de la división debe ponerse en las clases y la interacción de los objetos
- Dado que usaremos Python Notebooks, la subdivisión debe considerar adecuadamente la estructura de celdas.
- Recomendaciones/Exigencias:
 - Mantener una celda temáticamente coherente (familia de clases, clase y test, etc). No usar una única celda con todas las clases.
 - Partir con clases que modelan entidades evidentes, luego pasar a clases que manejen la coordinación.
 - Usar diagramas siempre que sea posible
 - **Siempre poner por escrito los supuestos y simplificaciones.**

¿Cómo podríamos partir entonces con la división del problema propuesto?

Escriba un programa que permita ejecutar una serie de procesos docentes de fin e inicio de semestre para una institución de educación superior.

Para especificar los prompts, es necesario dar más estructura que una conversación tradicional

Algunos de los elementos que siempre debe incluir un *prompt* (no solo para generar código):

- Propósito y requisitos: ¿qué debe hacer exactamente el código?
- Entradas y salidas: ¿qué datos recibirá y qué debe devolver? **Esto es fundamental para los métodos**
- Restricciones y supuestos: ¿hay alguna limitación que deba tener en cuenta?
- Hitos: ¿hay objetivos intermedios que deben cumplirse?
- Convenciones: ¿debe usarse alguna librería específica? ¿debe formatearse de alguna manera particular el código?

La siguiente celda contiene la definición de un algoritmo que asigna un estudiante a un curso y sección específico.

El prompt utilizado para genera el código fue el siguiente:

"Propósito: Escribe código orientado a objetos en Python que asigne un estudiante a un curso y sección específicos, validando que el curso exista, que la sección corresponda al curso y que haya cupos disponibles.

Entradas: id_estudiante (str), id_curso (str), id_seccion (str). Salida: Mensaje indicando si la inscripción fue exitosa o el motivo del rechazo.

Restricciones: No permitir inscripciones duplicadas ni en secciones sin cupo. Hito intermedio: Verificar existencia y consistencia de curso y sección antes de inscribir. Convenciones: El código es para un notebook de Python, por lo que se espera que se use el formato de celdas para ordenar el código."

```
[1] 1 class GestorInscripciones:
2     def __init__(self, cursos_repo):
3         self.cursos_repo = cursos_repo
4
5     def asignar_estudiante_a_seccion(self, id_estudiante: str, id_curso: str, id_seccion: str):
6         curso = self.cursos_repo.obtener_curso_por_id(id_curso)
7         if curso is None:
8             print(f"Error: El curso con ID '{id_curso}' no existe.")
9             return
10        seccion = curso.obtener_seccion_por_id(id_seccion)
11        if seccion is None:
12            print(f"Error: La sección '{id_seccion}' no corresponde al curso '{id_curso}'.")
13            return
14        if seccion.esta_inscrito(id_estudiante):
15            print(f"El estudiante {id_estudiante} ya está inscrito en la sección {id_seccion}.")
16            return
17        if not seccion.tiene_cupo():
18            print(f"No hay cupos disponibles en la sección {id_seccion}.")
19            return
20        seccion.inscribir_estudiante(id_estudiante)
21        print(f"Estudiante {id_estudiante} inscrito exitosamente en {id_curso} - {id_seccion}.")
22
```


Finalmente, todo el código (generado por IA o no) debe ser testeado de manera explícita

- Un test es simplemente código que muestra que otro código funciona como se espera. **Los casos que evalúa no son triviales.**
- No es necesario que haya un test por cada método (test unitario), ni tests para los tests, todo dependerá de la sofisticación y complejidad del código a evaluar (y del código del test).
- Tests tendrán puntaje explícito en las evaluaciones y deben ser convincentes.
- **Tip para uso de IA:** en vez de pedirle directamente a una IA que escriba cierto código, es mejor pedirle primero que escriba los tests que el código final debería superar. Luego, se le pide que implemente el código necesario para que esos tests se aprueben.

Para testear el código de la celda anterior, se generaron pruebas utilizando el siguiente prompt:

"Propósito: Genere código en Python que pruebe manualmente el método `GestorInscripciones.asignar_estudiante_a_seccion`, verificando los siguientes casos:

- Inscripción exitosa.
- Curso inexistente.
- Sección inválida (no corresponde al curso).
- Sin cupos disponibles.
- Estudiante ya inscrito.

Entradas y salidas:

- Entradas: distintos valores de `id_estudiante`, `id_curso` y `id_seccion`.
- Salidas: el programa debe mostrar en pantalla el nombre del caso, el resultado esperado y el resultado obtenido, de forma que un humano pueda verificar si el comportamiento es correcto.

Restricciones y supuestos:

- No usar librerías de testing automatizado.
- Asumir que `GestorInscripciones` y el resto de las clases necesarias ya existen.
- Simular o preparar los datos necesarios para cada caso de prueba usando las clases reales.

Hitos:

- Preparar los datos previos para cada escenario (configuración del repositorio, cursos y secciones).
- Ejecutar el método para cada caso de prueba.
- Mostrar en pantalla el resultado esperado y el obtenido para comparación visual.

Convenciones:

- El código debe estar organizado para ejecutarse en un notebook de Python, con una celda para la preparación de datos y otra para la ejecución de las pruebas.
- Los nombres de los casos y los mensajes deben ser claros y explícitos."

Cómo sigue la sesión de hoy

- Breve revisión de los ejercicios
- Trabajo en los ejercicios
- Entrega final del avance en repositorios y ticket de salida (17:10 a 17:30)

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación



IIC2115 - Programación como Herramienta para la Ingeniería

Fundamentos: POO y EDD

Profesor: Hans Löbel