ORIENTACIÓN A OBJETOS 1

Resumen teoría

Facultad de Informática

UNLP - LIFIA

2023

ÍNDICE

PROGRAMA O SISTEMA ORIENTADO A OBJETOS

¿Cómo es un software construido con objetos?

Un conjunto de objetos que colaboran enviándose mensajes. Todo computo ocurre “dentro” de los objetos

Los sistemas están compuestos (solamente) por un conjunto de objetos que colaboran para llevar a cabo sus responsabilidades.

Los objetos son responsables de:

- conocer sus propiedades
- conocer otros objetos (con los que colaboran)
- llevar a cabo ciertas acciones

ASPECTOS DE INTERÉS EN LA DEFINICIÓN

Algoritmos y datos ya no se piensan por separado •Cuando codificamos, describimos clases •Cuando se ejecuta el programa lo que tenemos son objetos que cooperan y que se crean dinámicamente durante la ejecución del programa

IMPACTO EN CÓMO PENSAMOS EL SOFTWARE

La estructura general cambia: en vez de una jerarquía:
Main/procedures/sub-procedures tenemos una red de “cosas” que se comunican

¿QUÉ ES UN OBJETO?

Es una abstracción de una entidad del dominio del problema. Ejemplos: Persona, Producto, Cuenta Bancaria, Auto, Plan de Estudios,….

Puede representar también conceptos del espacio de la solución (estructuras de datos, tipos “básicos”, archivos, ventanas, conexiones, iconos, adaptadores, …)

CARACTERÍSTICAS DE LOS OBJETOS

Un objeto tiene:

- Identidad: para distinguir un objeto de otro (independiente de sus propiedades)
- Conocimiento: En base a sus relaciones con otros objetos y su estado interno
- Comportamiento: Conjunto de mensajes que un objeto sabe responder

EL ESTADO INTERNO

El estado interno de un objeto determina su conocimiento.

El estado interno se mantiene en las variables de instancia (v.i.) del objeto.

Es privado del objeto. Ningún otro objeto puede accederlo. (¿Cuál es el impacto de esto?) El estado interno está dado por:

- Propiedades básicas (intrínsecas) del objeto
- Otros objetos con los cuales colabora para llevar a cabo sus responsabilidades.

COMPORTAMIENTO

Un objeto se define en términos de su comportamiento.

El comportamiento indica qué sabe hacer el objeto: Cuáles son sus responsabilidades.

Se especifica a través del conjunto de mensajes que el objeto sabe responder: protocolo

•La realización de cada mensaje (es decir, la manera en que un objeto responde a un mensaje) se especifica a través de un método. •Cuando un objeto recibe un mensaje responde activando el método asociado. •El que envía el mensaje delega en el receptor la manera de resolverlo, que es privada del objeto.

ENCAPSULAMIENTO

“Es la cualidad de los objetos de ocultar los detalles de implementación y su estado interno del mundo exterior”

Características:

- Esconde detalles de implementación.
- Protege el estado interno de los objetos.
- Un objeto sólo muestra su “cara visible” por medio de su protocolo.
- Los métodos y su estado quedan escondidos para cualquier otro objeto. Es el objeto quien decide qué se publica.
- Reduce el acoplamiento, facilita modularidad y reutilización

ENVÍO DE MENSAJE

- Para poder enviarle un mensaje a un objeto, hay que conocerlo.

- Al enviarle un mensaje a un objeto, éste responde activando el método asociado a ese mensaje (siempre y cuando exista).
- Como resultado del envío de un mensaje puede retornarse un objeto.

¿Cómo se especifica un mensaje?

- Nombre: correspondiente al protocolo del objeto receptor.
- Parámetros: información necesaria para resolver el mensaje.

(Cada lenguaje de programación propone una sintaxis particular para indicar el envío de un mensaje.)

MÉTODOS

¿Qué es un método?

Es la contraparte funcional del mensaje.

Expresa la forma de llevar a cabo la semántica propia de un mensaje particular (el cómo).

Un método puede realizar básicamente 3 cosas:

- Modificar el estado interno del objeto.
- Colaborar con otros objetos (enviándoles mensajes).
- Retornar y terminar

ENTRADA/SALIDA

Un objeto no debería realizar E/S (Imprimir en consola, recibir un input de teclado, etc)

En un sistema diseñado correctamente, un objeto de dominio (diseñado por un programador) no debería realizar ninguna operación vinculada a la interfaz(mostrar algo) o a la interacción (esperar un “input”).

De esta manera, separamos el programa de los dispositivos de E/S. ✔

METHOD LOOKUP

Cuando un objeto recibe un mensaje, se busca en su clase un método cuya firma se corresponda con el mensaje.

En un lenguaje dinámico, podría no encontrarlo (error en tiempo de ejecución)

En un lenguaje con tipado estático sabemos que lo entenderá (aunque no sabemos lo que hará)

FORMAS DE CONOCIMIENTO

Un objeto solo puede enviar mensajes a otros que conoce

Para que un objeto conozca a otro lo debe poder “nombrar” (debe tener una variable, con la referencia al otro objeto). Decimos que se establece una ligadura (binding) entre un nombre y un objeto.

Podemos identificar tres formas de conocimiento o tipos de relaciones entre objetos

- **Conocimiento Interno:** Variables de instancia.
- **Conocimiento Externo:** Parámetros.
- **Conocimiento Temporal:** Variables temporales.
- Existe una cuarta forma de conocimiento especial: las pseudo-variables (como “this” o “self”)

CLASES E INSTANCIA

Una clase es una descripción abstracta de un conjunto de objetos.

Las clases cumplen tres roles:

- Agrupan el comportamiento común a sus instancias. (métodos)
- Definen la forma de sus instancias (variables de instancia)
- Crean objetos que son instancia de ellas (cuando uno hace el New, se ejecuta el constructor)

En consecuencia todas las instancias de una clase se comportan de la misma manera.

Cada instancia mantendrá su propio estado interno.

ESPECIFICACIÓN DE CLASES

Las clases se especifican por medio de un nombre, el estado o estructura interna que tendrán sus instancias y los métodos asociados que definen el comportamiento

INSTANCIACIÓN

Es el **mecanismo de creación** de objetos.

Los objetos se instancian a partir de un molde. La clase funciona como molde.

Un nuevo objeto es una instancia de una clase.

Todas las instancias de una misma clase:

- Tendrá la misma estructura interna.
- Responderán al mismo protocolo (los mismos mensajes) de la misma manera (los mismos métodos).

Comúnmente se utiliza la palabra reservada new para instanciar nuevos objetos.

INICIALIZACIÓN

Para que un objeto esté listo para llevar a cabo sus responsabilidades hace falta inicializarlo. ¿De dónde sacamos esos valores iniciales?

Los objetos inicializan su estado interno en el constructor.

Pueden ser valores fijos, o pasados por parámetro

IDENTIDAD

Las variables son punteros a objetos

Más de una variable pueden apuntar a un mismo objeto

Para saber si dos variables apuntan al mismo objeto utilizó “==“

== es un operador, no puede definirse

IGUALDAD

Dos objetos pueden ser iguales

La igualdad se define en función del dominio

Para saber si dos objetos son iguales, uso “equals()”

Equals es un método, así que si se puede definir

THIS

this (o en algunos lenguajes self) es una “pseudo-variable”

- No puedo asignarle valor

- Toma valor automáticamente cuando un objeto comienza a ejecutar un método

- this hace referencia al objeto que ejecuta el método (al receptor del mensaje que resultó en la ejecución del método)

Se utiliza para:

- Descomponer métodos largos (top down)
- Reutilizar comportamiento repetido en varios métodos
- Aprovechar comportamiento heredado

TIPOS EN LENGUAJES OO

Tipo = Conjunto de firmas de operaciones/métodos (nombre, orden y tipos de los argumentos)

Decimos que un objeto (instancia de una clase) “es de un tipo” si ofrece el conjunto de operaciones definido por el tipo

Con eso en mente:

- Cada clase en Java define “explícitamente” un tipo (es un conjunto de firmas de operaciones)
- Cada instancia de una clase A (o de cualquier subclase) “es del tipo” definido por esa clase

Con respecto a herencia:

Donde espero un objeto de una clase A, acepto un objeto de cualquier subclase de A (lo opuesto no es cierto) (Por ejemplo, si espero que me pasen por parámetro una Lista, acepto tanto una ListaImplConArreglos, como una ListaEnlazada)

INTERFACES

Una clase define un tipo, y también implementa los métodos correspondientes y una variable tipada con una clase solo “acepta” instancias de esa clase

Una **interfaz** nos permite declarar tipos sin tener que ofrecer implementación (desacopla tipo e implementación). Es decir, se declaran sólo los métodos, NO las variables de instancia.

COLECCIONES

Es cuando un objeto que conoce a muchos …

Las relaciones de un objeto a muchos se implementan con colecciones

Decimos que un objeto conoce a muchos, pero en realidad conoce a una colección, que tiene referencias a esos muchos

Para modificar y explorar la relación, envío mensajes a la colección

POLIMORFISMO

Objetos de distintas clases son polimórficos con respecto a un **mensaje**, si todos lo entienden, aun cuando cada uno lo implemente de un modo diferente

Polimorfismo implica:

- Un mismo mensaje se puede enviar a objetos de distinta clase
- Objetos de distinta clase “podrían” ejecutar métodos diferentes en respuesta a un mismo mensaje

Por ejemplo, cuando dos clases Java implementan una interfaz, se vuelven polimórficas respecto a los métodos de la interfaz

Un polimorfismo bien aplicado:

- Permite repartir mejor las responsabilidades (delegar)
- Desacopla objetos y mejora la cohesión (cada cual hace lo suyo) • Concentra cambios (reduce el impacto de los cambios)
- Permite extender sin modificar (agregando nuevos objetos)
- Lleva a código más genérico y objetos reusables
- Nos permite programar por protocolo, no por implementación

HERENCIA

Mecanismo que permite a una clase “heredar” estructura y comportamiento de otra clase • Es una estrategia de reuso de código

Es una estrategia para reúso de conceptos

Es una característica transitiva

(Transitivo = Que pasa y se transfiere de uno a otro. Por definición, requiere la existencia de al menos un par)

La prueba “es un”…

Preguntarse “es-un” es la regla para identificar usos adecuados de herencia

Si suena bien en el lenguaje del dominio, es probable que sea un uso adecuado

METHOD LOOKUP CON HERENCIA

Cuando un objeto recibe un mensaje, se busca en su clase un método cuya firma se corresponda con el mensaje. Si no lo encuentra, sigue buscando en la superclase de su clase, y en la superclase de esta…

SOBREESCRITURA DE MÉTODOS (OVERRIDING)

La búsqueda en la cadena de superclases termina tan pronto encuentro un método cuya firma coincide con la que busco.

Si un objeto heredaba un método con exactamente la misma firma que uno que tiene definido, el heredado queda “oculto” (redefinir / override)

No es algo que ocurra con frecuencia.

EXTENDER MÉTODOS

Con la pseudovariable “super” podemos extender el comportamiento heredado (reimplementar un método e incluir el comportamiento que se heredaba para él)

En un método, “super” y “this” **hacen referencia** al objeto que lo ejecuta (al receptor del mensaje). Lo único que cambia es que utilizar “super” es la forma de hacer el method lookup, que comienza a buscar desde la clase inmediatamente superior.

No confundir con el método Super().

Este se usa en los constructores de la subclase para reutilizar el comportamiento de la clase superior.

ESPECIALIZAR

Especializar: crear una subclase especializando una clase existente

CLASE ABSTRACTA

Una clase abstracta captura comportamiento y estructura que será común a otras clases.

- Se pueden declarar variables de instancia, constructor y métodos.
- Una clase abstracta no se puede instanciar (no modela algo completo)
- Seguramente será especializada.
- Puede declarar comportamiento abstracto y utilizarlo para implementar comportamiento concreto

GENERALIZAR

Generalizar: Introducir una superclase que abstrae aspectos comunes a otras – suele resultar en una clase abstracta

DIAGRAMA DE CLASES:

Un diagrama de clases en Lenguaje Unificado de Modelado (UML) es un tipo de diagrama de estructura estática que describe la estructura de un sistema mostrando las clases del sistema, sus atributos, operaciones, y las relaciones entre los objetos.

DIAGRAMA DE OBJETOS:

En UML, diagrama que muestra una vista completa o parcial de los objetos de un sistema en un instante de ejecución específico

DIAGRAMA DE PAQUETES

Un diagrama de paquetes en el Lenguaje Unificado de Modelado representa las dependencias entre los paquetes que componen un modelo. Es decir, muestra cómo un sistema está dividido en agrupaciones lógicas y las dependencias entre esas agrupaciones

DIAGRAMA DE SECUENCIA

Un diagrama de secuencia muestra la interacción de un conjunto de objetos en una aplicación a través del tiempo y se modela para cada caso de uso. A menudo es útil para complementar a un diagrama de clases, pues el diagrama de secuencia se podría describir de manera informal como "el diagrama de clases en movimiento", por lo que ambos deben estar relacionados entre sí (mismas clases, métodos, atributos...).

TESTING

¿Qué es testear?

Asegurarse de que el programa:

- hace lo que se espera
- lo hace como se espera
- no falla

TEST DE UNIDAD

Test que asegura que la unidad mínima de nuestro programa funciona correctamente, y aislada de otras unidades (En nuestro caso, la unidad de test es el método)

Testear un método es confirmar que el mismo acepta el rango esperado de entradas, y que retorna el valor esperado en cada caso

- tengo en cuenta parámetros
- estado del objeto antes de ejecutar el método
- objeto que retorna el método
- estado del objeto al concluir la ejecución del método

TEST AUTOMATIZADOS

Se utiliza software para guiar la ejecución de los tests y controlar los resultados

Requiere que diseñemos, programemos y mantengamos programas “tests”

En nuestro casos serán objetos

Suele basarse en herramientas que resuelven gran parte del trabajo

Una vez escritos, los puedo reproducir a costo mínimo, cuando quiera

Los tests son “parte del software” (y un indicador de su calidad)

INDEPENDENCIA ENTRE TEST

No puedo asumir que otro test se ejecutó antes o se ejecutará después del que estoy escribiendo

¿POR QUÉ Y CUANDO TESTEAR?

- Testeamos para encontrar bugs
- Testeamos con un propósito (buscamos algo)
- Pensamos por qué testear algo y con qué nivel queremos hacerlo
- Testeamos temprano y frecuentemente
- Testeo tanto como sea el riesgo del artefacto

TEST DE PARTICIONES EQUIVALENTES

Partición de equivalencia: conjunto de casos que prueban lo mismo o revelan el mismo bug. Asumo que si un ejemplo de una partición pasa el test, los otros también lo harán, ergo elijo uno.

Si se trata de valores en un rango, tomo un caso dentro y uno por fuera en cada lado del rango. Si se trata de casos en un conjunto, tomo un caso que pertenezca al conjunto y uno que no.

TEST CON VALORES DE BORDE

Los errores ocurren con frecuencia en los límites y ahí es donde los vamos a buscar Intentamos identificar bordes en nuestras particiones de equivalencia y elegimos esos valores

ANÁLISIS Y DISEÑO

Análisis: El análisis pone énfasis en una investigación del problema y los requisitos, en lugar de ponerlo en la solución.

Diseño: El diseño pone énfasis en una solución conceptual que satisface los requisitos, en lugar de ponerlo en la implementación

RESPONSABILIDADES DE LOS OBJETOS

Conocer:

- Conocer sus datos privados encapsulados
- Conocer sus objetos relacionados
- Conocer cosas derivables o calculables

Hacer:

- Hacer algo él mismo

- Iniciar una acción en otros objetos
- Controlar o coordinar actividades de otros objetos

HEURÍSTICAS PARA ASIGNACIÓN DE RESPONSABILIDADES

Significado Heurística:

> 1 Conjunto de técnicas empleadas para resolver problemas
>
> 2 Técnica de la indagación y del descubrimiento

La habilidad para asignar responsabilidades es extremadamente importante en el diseño.

La asignación de responsabilidades generalmente ocurre durante la creación de diagramas de interacción.

EXPERTO EN INFORMACIÓN

Descripción: Asignar una responsabilidad al experto en información (la clase que tiene la información necesaria para realizar la responsabilidad).

Expresa la intuición de que los objetos hacen cosas relacionadas con la información que tienen.

Para cumplir con su responsabilidad, un objeto puede requerir de información que se encuentra dispersa en diferentes clases a expertos en información “parcial”.

Ejemplo:

> ¿Quién tiene la responsabilidad de conocer el monto total de una compra?… La compra.
>
> Entonces: LineaDeVenta es responsable de conocer el subtotal por cada ítem EspecificaciónDelProducto es responsable de conocer el precio del ítem.

CREADOR

Descripción: asignar a la clase B la responsabilidad de crear una instancia de la clase A si:

- B contiene objetos A (una colección en una variable de instancia).
- B registra instancias de A.
- B tiene los datos necesarios para inicializar objetos A.
- B usa a objetos A en forma exclusiva.

Ejemplo:

> ¿Quién debe ser responsable de crear una LineaDeVenta?… La compra La intención del Creador es determinar una clase que necesite conectarse al objeto creado en alguna situación. Eligiéndolo como el creador se favorece el bajo acoplamiento

CONTROLADOR

Descripción: asignar la responsabilidad de manejar eventos del sistema a una clase que representa: El sistema global, dispositivo o subsistema

Ejemplo:

> ¿Quién debe ser el controlador de los eventos ingresarLibro o finalizarCompra?… ManejadorCompras, Librería.
>
> La intención del Controlador es encontrar manejadores de los eventos del sistema, sin recargar de responsabilidad a un solo objeto y manteniendo alta cohesión.

BAJO ACOPLAMIENTO

Descripción: asignar responsabilidades de manera que el acoplamiento permanezca lo más bajo posible. El acoplamiento es una medida de dependencia de un objeto con otros. Es bajo si mantiene pocas relaciones con otros objetos.

El alto acoplamiento dificulta el entendimiento y complica la propagación de cambios en el diseño.

No se puede considerar de manera aislada a otras heurísticas, sino que debe incluirse como principio de diseño que influye en la elección de la asignación de responsabilidad.

ALTA COHESIÓN

Descripción: asignar responsabilidades de manera que la cohesión permanezca lo más fuerte posible. La cohesión es una medida de la fuerza con la que se relacionan las responsabilidades de un objeto, y la cantidad de ellas. Es decir, un objeto no debe tener 2 millones de responsabilidades. Y las cosas que tiene que hacer, deben estar relacionadas entre sí (que tenga sentido que estén juntas)

Ventaja: clases más fáciles de mantener, entender y reutilizar.

El nivel de cohesión no se puede considerar de manera aislada a otras responsabilidades y otras heurísticas, como Experto y Bajo Acoplamiento

POLIMORFISMO

Descripción: cuando el comportamiento varía según el tipo, asigne la responsabilidad a los tipos/las clases para las que varía el comportamiento.

Ejemplo:

> El sistema de venta de libros debe soportar distintas bonificaciones de pago con tarjeta de crédito. (ya visto previamente)… Como la bonificación del pago varía según el tipo de tarjeta, deberíamos asignarle la responsabilidad de la bonificación a los distintos tipos de tarjeta.

Nos permite sustituir objetos que tienen idéntica interfaz.

“NO HABLES CON EXTRAÑOS”

Descripción: Evite diseñar objetos que recorren largos caminos de estructura y envían mensajes (hablan) a objetos distantes o indirectos (extraños).

Dentro de un método sólo pueden enviarse mensajes a objetos conocidos:

- Self/this
- un parámetro del método
- un objeto que esté asociado a self/this
- un miembro de una colección que sea atributo de self/this
- un objeto creado dentro del método

Los demás objetos son extraños (strangers)

REUSO DE CÓDIGO (Herencia vs Composición)

Las clases y los objetos creados mediante herencia están estrechamente acoplados ya que cambiar algo en la superclase afecta directamente a la/las subclases.

Las clases y los objetos creados a través de la composición están débilmente acoplados, lo que significa que se pueden cambiar más fácilmente los componentes sin afectar el objeto contenedor

HERENCIA (Relaciones “Es un”)

Herencia total: debo conocer todo el código que se hereda -> Reutilización de Caja Blanca

Todo lo implementado heredado (de toda la cadena de herencias) debe ser util.

Usualmente debemos redefinir o anular métodos heredados

Los cambios en la superclase se propagan automáticamente a las subclases

¿Herencia de Estructura vs. Herencia de comportamiento?

Es preferible definir la herencia por **comportamiento común**

Es útil para extender la funcionalidad del dominio de aplicación

COMPOSICIÓN (Relaciones “tiene un” o “usa un”)

Los objetos se componen en forma Dinámica -> Reutilización de Caja Negra

Los objetos pueden reutilizarse a través de su interfaz (sin conocer el código)

A través de las relaciones de composición se pueden delegar responsabilidades entre los objetos.

PRINCIPIOS SOLID

Relacionados a las HAR, para un buen estilo de DOO.

Promueven Alta Cohesión y Bajo Acoplamiento.

S SRP: The Single-Responsibility Principle

Principio de Responsabilidad única. Más de lo mismo, alta cohesión.

Una clase debería cambiar por una sola razón. Debería ser responsable de únicamente una tarea, y ser modificada por una sola razón (alta cohesión)

O OCP: The Open-Closed Principle

Entidades de software (clases, módulos, funciones, etc.) deberían ser “abiertas” para extensión, y “cerradas” para modificación.

Abierto a extensión: ser capaz de añadir nuevas funcionalidades.

Cerrado a modificación: al añadir la nueva funcionalidad no se debe cambiar el diseño existente. No tocar lo que ya está hecho.

L LSP: The Liskov Substitution Principle

Los objetos de un programa deben ser intercambiables por instancias de sus subtipos sin alterar el correcto funcionamiento del programa.

Es decir, que si el programa utiliza una clase (clase A), y ésta es extendida (clases B, C, D, etc…) el programa tiene que poder utilizar cualquiera de sus subclases y seguir siendo válido. Uso correcto de herencia (Es-un) y polimorfismo.

I ISP: The Interface-Segregation Principle

Las clases que tienen interfaces “voluminosas” son clases cuyas interfaces no son cohesivas.

Las clases no deberían verse forzadas a depender de interfaces que no utilizan. Cuando creamos interfaces (protocolos) para definir comportamientos, las clases que las implementan, no deben estar forzadas a incluir métodos que no va a utilizar.

D DIP: The Dependency-Inversion Principle

a. Los módulos de alto nivel de abstracción no deben depender de los de bajo nivel.

b. Las abstracciones no deben depender de detalles. Los detalles deben depender de las abstracciones

Ser capaz de «invertir» una dependencia es lo mismo que ser capaz de «intercambiar» una implementación concreta por otra implementación concreta cualquiera, respecto a la misma abstracción.

- **Módulos de alto nivel:** se refieren a los objetos que definen qué es y qué hace el sistema.

- **Módulos de bajo nivel:** no están directamente relacionados con la lógica de negocio del programa (no definen el dominio). Por ejemplo, el mecanismo de persistencia o el acceso a red .
- **Abstracciones:** se refieren a protocolos (o interfaces) o clases abstractas.
- **Detalles:** son las implementaciones concretas, (cuál mecanismo de persistencia, etc).

SMALLTALK

Es un lenguaje OO puro – todo es un objeto (¡ incluso las clases !)

Tipado dinámicamente

Propone una estrategia exploratoria (construccionista) al desarrollo de software

El ambiente es tan importante como el lenguaje

- Está implementado en Smalltalk
- Ricas librerías de clases (fuentes de inspiración y ejemplos)
- Todo su código fuente disponible y modificable
- Tiene su propio compilador, debugger, editor, inspector, perfilador, etc.
- Es extensible

Sintaxis minimalista (con sustento en su foco educativo)

Fuente de inspiración de casi todo lo que vino después (en OO)

En Smalltalk Smalltalk las clases son objetos:

Hay dos tipos de objetos: los que pueden crear instancias (de sí mismos), y los que no.

- A los primeros les llamamos clases.

Si las clases entienden mensajes, tienen su propio conocimiento y comportamiento

- ¿Dónde se especifica su estructura y comportamiento? ¿En otra clase?

Esto (El metamodelo de Smalltalk) en uno de sus aspectos más interesantes y desafiantes

Metamodelo:

- Las clases son instancias de una clase también (su metaclase).
- Por cada clase hay una metaclase (se crean juntas).
- SmallInteger es instancia de “SmallInteger class”
- Las metaclases son instancias de la clase Metaclass •

  Por ejemplo:“SmallInteger class” es instancia de Metaclas

JAVASCRIPT (ECMASCRIPT)

Lenguaje de propósito general

Dinámico

Basado en objetos (con base en prototipos en lugar de clases)

Multiparadigma • Se adapta a una amplia variedad de estilos de programación

Pensado originalmente para scripting de páginas web (Con una fuerte adopción en el lado del servidor (NodeJS))

PROTOTIPOS

En Javascript no tengo clases

La forma más simple de crear un objeto es mediante la notación literal (estilo JSON)

Cada objeto puede tener su propio comportamiento (métodos)

Los objetos heredan comportamiento y estado de otros (sus prototipos)

Cualquier objeto puede servir como prototipo de otro

Puedo cambiar el prototipo de un objeto (y así su comportamiento y estado)

Termino armando cadenas de delegación