

Resumen Práctica - Objetos 1

1

MALOS OLORES DE DISEÑO

- ENVIDIA DE ATRIBUTOS → cuando un objeto le pide cosas a otros objetos para hacer algo él mismo

PARA
EVITARLO

↳ la tarea la debe hacer el objeto que tiene las cosas que se necesitan → delegárselo a él

- CLASE DIOS → cuando una clase requiere todo y las demás son animadas (método // también haya envidia)

PARA
EVITARLO

↳ ver otros objetos que puedan aparecer o ya lo conozcan y que puedan ser responsables de algo

- CÓDIGO DUPLICADO → cuando copio y pego código en varios lugares

PARA
EVITARLO

↳ podría generalizar el comportamiento en una clase y heredarlo

→ llevarlo a otro objeto y usarlo por composición
→ extraerlo en un método en la misma clase y reusarlo

- CLASE LARGA → cuando tengo una clase muy grande en comparación del resto

PARA
EVITARLO

→ delegar algo en otros objetos que conoce
→ pensirla como una composición de varios objetos

- MÉTODO LARGO → cuando un método tiene más de 20 renglones.

(si DEBO incluir comentarios en medio de un método).

PARA
EVITARLO

→ identificar partes que podrían ser comportamientos individuales.

→ llevar cada parte a un nuevo método
→ utilizar un BUEN nombre para el método
→ llamarlo (enviar mensajes) a this.

• OBJETOS QUE CONOCEN EL ID DE OTRO

→ Nunca relacionar objetos por medio de claves o ids!

PARA

wando un objeto se relaciona con otro, lo hace con una referencia (nunca conoce su id)

• ESO DEBERÍA SER UN OBJETO (OBSERVACIÓN POR LOS PRIMITIVOS)

→ A veces modelamos como Strings o números (primitivos) que deberían ser objetos

PARA

pensar si eso que estoy modelando como un String o un número (primitivo) no debería ser modelado como una clase específica

• SWITCH STATEMENTS → cuando se usan if/case/switch/if anidados para identificar de qué forma hay algo

PARA

EVITARLO → aplico polimorfismo

• VARIABLES DE INSTANCIA QUE DEBERÍAN SER TEMPORALES

→ Si una variable de instancia deja de tener sentido/en el último momento de la vida del objeto, entonces es probable que sea temporal o sea responsabilidad de otro

PARA

EVITARLO → pensar si esa variable es un atributo del objeto o es algo temporal de un método

• ROMPER ENCAPSULAMIENTO → MUY MALO. (): acceder a variables de instancia de otros objetos / setters / getters innecesarios para todas las variables

PARA

EVITARLO → pensar si esa variable es un atributo del objeto o es algo temporal de un método

→ agregar setters / getters cuando sea necesario, nunca modificar una colección que no es nuestra, delegar la tarea a los que tienen la info. que se necesita

Resumen Práctica - Objeto 1

2

- CLASE DE DATOS o CLASE ANÉMICA → una clase que parece un registro de datos debería dar mala impresión
PARA EVITARLO → asegurarse que no hay comportamiento en el sistema que debería estar haciendo esa clase, y lo hace otro objeto
- NO ES-UN → cuando no se cumple la regla es-un de la clase hija a la superclase
PARA EVITARLO → la respuesta siempre debe ser Si cuando defino una subclase y me pregunto ¿es-un? en relación a la superclase
- NO QUIERO MI HERENCIA → cuando un método redefinido heredado pero hace algo total / ≠ peor caso: redifinir un método heredado indicando finalmente que no heredará
PARA EVITARLO → pensar si no puedo reorganizar la jerarquía de clases para que ninguna clase herede comportamientos que no quiere
- REINVENTANDO LA RUEDA → mal idea cuando defino comportamiento que ya está programado en algún lado (algún objeto ya sabe hacer eso)
(ej: implementar compareTo(), equals(), hashCode() que x y los streams ya saben hacer)
PARA EVITARLO → utilizar comportamiento que ya fue definido
→ invento clases y protocolos que ofrecen las librerías de objetos

ESTILO DE PROGRAMACIÓN

- OFRECER CONSTRUCCIONES → garantizar una buena iniciación
- NOMBRE DE MENSAJE QUE REVELA LA INTENCIÓN
 - ↳ que el nombre del msg comunigue lo que se quiere hacer (no cómo)
- DELEGACIÓN A THIS → permite descomponer un método en partes que el mismo objeto resuelve
 - cada método hace una cosa. Su nombre indica lo que hace (nombre corto)
 - permite que la subclase redefina/extienda sólo un paso
- MÉTODOS CORTOS → tener métodos cortos que utilicen nombres de mensajes que revelan la intención
- CADA COSA SE HACE UNA SOLA VEZ → utilizar comportamiento de colecciones y/o otros objetos
- LOS NOMBRES DE LAS VARIABLES DEBEN INDICAR SU ROL
 - ↳ elegir nombres de variables para que quede claro qué rol cumplen en el método/clase.
 - comienzan con minúscula y sintaxis de camelCase
- PIENSA BIEN LOS NOMBRES DE LAS CLASES
 - ↳ siempre empiezan con mayúscula y singular. Sintaxis de camelCase capitalizada.

Resumen Práctico - Objetos 1

3

STREAMS

ClienteDeCorreo

```
• public Email buscarEmail (String texto){  
    return this.carpetas.stream()  
        .map (carpeta → carpeta.buscarEmail(texto))  
        .filter (Object::NonNull)  
        .orElse (null);  
}
```

Carpeta

```
• public Email buscarEmail (String texto){  
    return this.emails.stream()  
        .filter (e → e.contains (texto) == true)  
        .findFirst()  
        .orElse (null);  
}
```

Email

```
• public boolean contiene (String texto){  
    return this.titulos.contains (texto) || this.cuerpos.  
        contains (texto);  
}
```

.getLast()
.size()
.sum()
.count()
.findFirst()
.findAny()

.allMatch ()
.filter()
.mapToDouble()
.stream().collect(Collectors.groupingBy(e → e.get(category),
 Collectors.Counting()));
.forEach ().

LOCALDATE
 • `getDayOfWeek().toString().equals("sunday")`
CHRONOUNIT
`(int) ChronoUnit.DAYS.between(Fecha, Fecha2);`
`(int) -1 'YEARS' within 1998-01-01T00:00:00Z`

DATELAPSE
`private LocalDate to;`
`private LocalDate from;`
`+ getTo(): LocalDate`
`+ getFrom(): LocalDate`
`+ sizeInDays(): int`
`+ includesDate(LocalDate fecha): Boolean`
`+ overlaps(Datelapse periodo): Boolean`
`+estaEnCurso(): Boolean`
`+estaEntre(Datelapse periodo): Boolean`

SET / LINKED HASH SET

NO acepta valores repetidos mantiene el orden de agregación

MHASHSET → NO mantiene orden de agregación

TREESET → ordena según un comparador

BAG → Utiliza `HashMap<String, Integer>`

- ↓ ↓
- `add(element)` • `put(element, value)`
- `occurrences(element)` • `get(element)`
- `moreOccurrences(element)` • `remove(element)` / `put(element, get(element)-1)`

en oo1 TEST AUTOMATIZADO (DE UNIDAD)

↳ Son 4 tipos de TEST FUNCIONALES.

se utiliza software (escribimos código)

4

TESTING → para asegurarse que el programa haga lo que se espera y no falle

→ NO se testean Interfaces y Clases Abstractas

→ NO se testean getters, setters ni constructores

→ una clase Test por cada clase Testable

→ empezar con los Test de unidad y después seguir con los tests de integración

test de unidad → asegurarse que la unidad mínima de nuestro programa funcione correctamente

→ acepta参ámetros y devuelve valor esperado

BUSCAR / IDENTIFICAR PARTICIONES Y VALORES DE BORDE

RON CADA UME

→ identificar métodos a testear

→ identificar particiones / bordes para cada método

public class NombreClaseTest {

// Declaraciones.

NombreClase n;

// Instanciaciones

@BeforeEach

void setUp() {

n = NombreClase(..., ..., ...);

}

// Tests

@Test

void metodoTgt() {

} ... n tests.

specifica particiones y valores

de borde

→ xUnit (nombre genérico)

JUNIT → framework (en Java) para automatizar la ejecución de tests de unidad

test automatizado

↳ programa que se va a asegurar que la clase donde están esos métodos funcione correctamente

ASSERTS

- assertEquals(1, 1);
- assertNotEquals(1, 2);
- assertNotNull(object);
- assertNull(null);
- assertSame(object, object);
- assertTrue(true);
- assertFalse(false);

(condición)? true : false;

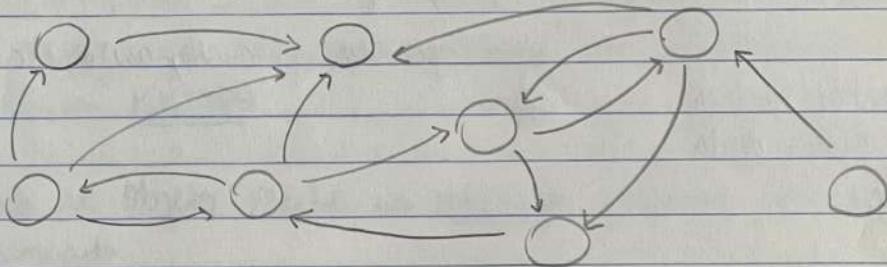
Resumen Teoría - Objetos 1

Programa o Sistema orientado a Objetos

¿Cómo es un software construido con objetos?

↓
conjunto de objetos que colaboran enviándose mensajes.

Todo COMUNTO OCURRE "DENTRO" DE LOS OBJETOS



La clave del éxito es poder agregar nueva funcionalidad (no prevista original), reemplazar o modificar objetos y que el sistema "no se entere" ni se rompa (ej: integración WhatsApp y Facebook mess.)

→ los sistemas están compuestos (Sola!!) por un conjunto de objetos que colaboran para llevar a cabo sus responsabilidades.

RESPONSABILIDADES DE LOS OBJETOS

- Conocer sus propiedades
- Conocer otros objetos (con los que colaboran)
- Llevar a cabo ciertas acciones

ASPECTOS DE INTERÉS:

- No hay un objeto main
- cuando codificamos, programamos clases
- una jerarquía de clases Nº indica lo mismo que la jerarquía top-down
- cuando se ejecuta el programa lo que tenemos son objetos que cooperan y que se crean dinámicamente durante la ejecución del programa

Identificar cuáles son las clases

y dentro de las clases → cuáles son los métodos

OBJETO → abstracción de una entidad del dominio del problema

↓
puede representar también

conceptos del espacio de la
solución

↓
persona, producto, auto, etc.

OBJETO → identidad → para distinguir un objeto de otro

conocimiento → en base a sus relaciones con otros objetos y
su estado interno

comportamiento → conjunto de mensajes que un objeto sabe responder

ESTADO INTERNO

→ determina su conocimiento

→ dado por:

• propiedades básicas del objeto

• objetos con los que colabora para llevar a cabo sus responsabilidades

→ se mantiene en las variables de instancia del objeto

→ es PRIMARIO del objeto

Un objeto se define a través de su comportamiento

→ qué sabe hacer

se especifica a través del

conjunto de msgs que el

objeto sabe responder: protocolo

↓
responsabilidades

Resumen Teoría - Objetos 1

2

VARIABLES DE INSTANCIA

↳ en general son REFERENCIAS (punteros) a otros objetos con los cuales el objeto colabora

→ algunos pueden ser atributos básicos

COMPORTAMIENTO - IMPLEMENTACIÓN

• la realización de cada mensaje se especifica a través de un MÉTODO

• cuando un objeto recibe un mensaje responde activando el método asociado

• el que envía el msg delega en el receptor la manera de resolverlo, que es privada del objeto

ENVÍO DE UN MENSAJE

• para poder enviarle un msg a un objeto, hay que conocerlo.

• al enviarle un msg a un objeto, éste responde activando el método asociado a ese msg (siempre y cuando sea posible)

• como resultado del envío de un msg puede retornarse un objeto

ESPECIFICACIÓN DE UN MENSAJE

• Nombre: correspondiente al protocolo del objeto receptor

• Parámetros: información necesaria para resolver el mensaje (cada lenguaje de programación propone una sintaxis particular)

ej: cuenta.depositar (cantidad)

MÉTODO → contraparte funcional del objeto
expresa la forma de llevar a cabo la semántica propia de un mensaje particular (el cómo)

Puede → modificar el estado interno del objeto
colaborar con otros objetos (enviándoles msgs)
retornar y terminar

En un sistema diseñado correcto, un objeto NO debería realizar/leer/imprimir operaciones de entrada y salida

FORMAS DE CONOCIMIENTO

→ para que un objeto conozca a otro lo debe poder "nombrar"
→ decimos que se establece una ligadura (binding) entre un nombre y un objeto

- FORMAS DE CONOCIMIENTO / TIPOS DE RELACIONES ENTRE OBJETOS
 - CONOCIMIENTO INTERNO → variables de instancia
 - CONOCIMIENTO EXTERNO → parámetros
 - CONOCIMIENTO TEMPORAL → variables temporales
 - CONOCIMIENTO "ESPECIAL" → pseudo-variables ("this" o "self")

ENCAPSULAMIENTO → cualidad de los objetos de ocultar los detalles de implementación y su estado interno

- ↓
- 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 otro objeto. El objeto quien decide qué se publica
 - facilita modularidad y reutilización

Resumen Teoría - Objetos 1

3

CLASES E INSTANCIAS

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

→ Las clases cumplen 3 roles:

- agrupan el comportamiento común a sus instancias
- definen la forma de sus instancias
- crean objetos que son instancias de ellos

• todos los 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 estructura o estructura interna que tendrán sus instancias y los métodos asociados que definen el comportamiento.

Instanciación

- mecanismo de creación de objetos
- objetos se instancian a partir de un molde
- la **clase** funciona como molde
- un objeto es una instancia de una clase
- TODAS las instancias de una misma clase:
 - ↳ tendrán la misma estructura interna
 - ↳ responderán al mismo protocolo (mismos msgs) de la misma manera (mismos métodos)

Iniciación

• para que un objeto esté listo para llevar a cabo sus responsabilidades hace falta inicializarlo

↳ darle valor a sus variables (constructores)

un programa en objetos cuando se empieza a ejecutar es un grafo.

RELACIONES ENTRE OBJETOS

- un objeto conoce a otro porque
 - es su responsabilidad mantener a ese otro objeto en el sistema
 - necesita delegarle trabajo (enviarle msjs)
- un objeto conoce a otro cuando
 - tiene una referencia en una var. de instancia (rel. duradera)
 - le llega una referencia como parámetro (rel. temporal)
 - lo crea (rel. temporal / duradera)
 - lo obtiene enviando msjs a otros que conoce (rel. temporal)

this (o en algunos den grajes "self")

- es una pseudo-variable
- no puede asignarle valor
- toma valor automáticamente cuando un objeto comienza a ejecutar un método
- hace referencia al objeto que ejecuta el método
- se utiliza para
 - descomponer métodos largos
 - reutilizar comportamiento repetido en varios métodos
 - aprovechar comportamiento heredado
 - pasar una referencia para que otros puedan enviarlos msjs

BINDING DINÁMICO

- ↳ cuando un objeto recibe un msj va a su clase a buscar el método que se llama y lo ejecuta (si no lo encuentra, busca en su clase padre, etc)

las variables son referencias (puntan a objetos)

↳ comparo utilizando "=="

Resumen Teoría - Objetos 1

RELACIONES ENTRE OBJETOS Y CHEQUEOS DE TIPOS

→ Java es un lenguaje, estático//, fuerte// tipado

↳ debemos indicar el tipo de todos los variables

↳ el compilador chequea la correctitud de nuestro programa respecto a tipos

→ Se asegura que no enviamos msgs a objetos que no los entienden

TIPOS EN LENGUAJES OO

- Tipo → conjunto de primos de operaciones / métodos (membre, orden y tipos de los argumentos)
- cada clase en Java define "explícita// " un tipo (es un conjunto de primos de operaciones)

INTERFAZES

- una clase define un tipo, y tmb implementa los métodos correspondientes
- una variable tipada con una clase solo "acepta" instancias de esa clase *
- una interfaz nos permite declarar tipos sin tener que ofrecer implementación
- puedo usar interfaces como tipos de Variables
- las clases deben declarar explícita// que interfaces implementan
 - ↳ el compilador chequea que la clase las implemente

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 referencia a esos muchos
- para modificar y explorar la relación, envío msgs a la colección

Un objeto NUNCA cambia de clase

* (y herencia...)

POLIMORFISMO

- Objetos de distintas clases son polimórficos con respecto a un msg, si todos lo entienden, aun cuando cada uno lo implemente de un modo diferente
- Polimorfismo implica:
- un mismo msg se puede enviar a objetos de distinta clase
 - objetos de distinta clase "polífan" ejecutar métodos diferentes en respuesta a un mismo msg
- Cuando dos clases JAVA implementan una interfaz, se vuelven polimórficos respecto a los métodos de la interfaz

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
- Reuse de código
 - Reuse de conceptos/definiciones.
- En JAVA solo tenemos herencia simple

ES-UN

- presentarse es-un es la regla para identificar usos adecuados de herencia

METHOD LOOK UP CON HERENCIA

- Cuando un objeto recibe un msg, se busca en su clase un método cuya firma se corresponda con el msg. Si no lo encuentra, busca en la Superclase, y así sucesivamente

Resumen Teoría - Objetos 1

5

Sobre escribir métodos (overriding)

- Sobre escribir método heredado → no es recomendado

SUPER

- "pseudo-variable" (toma valor automática) cuando un objeto emplea a ejecutar un método
- cuando super recibe un msg, la búsqueda de métodos comienza en la clase inmediata superior a aquella donde está definido el método que envía el msg
- Super en un constructor para ejecutar constructor de clase padre

Especializar

- Crear una subclase especializando una clase existente

CLASE ABSTRACTA

- Captura comportamiento y estructura que será común a otras clases
- No tiene instancias (no modela algo completo)
- Seguramente será especializada
- Puede declarar comportamiento abstracto y utilizarlo para implementar comportamientos concretos

GENERALIZAR

- Introducir una superclase que abstraiga aspectos comunes a otras → Se resulta en una clase abstracta

SITUACIONES DE USO DE HERENCIA

- Subclasicar para especializar → Subclase extiende métodos para especializarse
contra clases concretas
- Herencia para especificar → Superclase: generalización de métodos concretos y abstractos (clase abstracta)
Subclase implementa los métodos abstractos.
- Subclasicar para extender → Subclase agrega nuevos métodos.

2

1. abstracto - abstract

CLASES ABSTRACTAS E INTERFACES

→ una clase abstracta es una clase

- puede usarse como tipo
- puede o no tener métodos abstractos
- ofrece implementación a algunos métodos
- sus métodos concretos pueden depender de sus métodos abstractos

→ Una interfaz define un tipo

- sirve como "contrato"
- puede entender a otras

→ Se pueden implementar muchas interfaces pero solo se puede heredar de una clase

COLECCIONES

→ Nunca modifiques una colección que obtuve de otro objeto

→ Cada objeto es responsable de mantener los invariantes de sus colecciones

→ Solo el dueño de la colección puede modificarla

→ Recordar que una colección puede cambiar luego de que la obtengo

LIBRERÍA / FRANWERK DE COLECCIONES

→ todos los lenguajes OO ofrecen librerías de colecciones

→ las colecciones admiten, generalmente, contenido heterogéneo en términos de clase, pero homogéneo en términos de comportamiento

→ colecciones de JAVA:

- interfaces
- clases abstractas
- clases concretas
- algoritmos útiles

Q: LIST (admite duplicados, indexados de 0 en adelante)

SET (NO admite duplicados, elementos NO indexados)

MAP (elementos asociados a claves)

QUEUE (manejó el orden de recuperación de objetos (LIFO, FIFO, prioridad, etc))

Resumen Teoría - Objetos 1

6

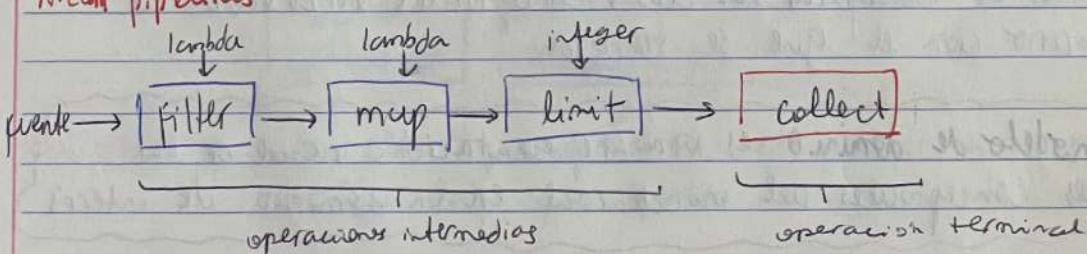
ITERATION (lo entienden todos los colecciones)

- proporciona una manera de recorrer (o iterar) sobre los elementos de una colección de forma secuencial sin esperar su representación interna

STREAM

- código más fácil de entender y mantener
- cada operación produce un resultado, pero NO modifica la lista

Stream pipelines



UML

- lenguaje de modelado visual que nos permite
 - especificar
 - visualizar
 - construir
 - documentar
- } ... artefactos de un sistema de software

Lenguaje UML

- ↳ permite capturar decisiones y conocimientos

Diagramas de estructura

- de clases
- de paquetes
- de componentes
- de objetos
- de despliegue

Diagramas de comportamiento

- de casos de uso
- de interacción
- de máquinas de estados
- de actividades

MODELO DE DOMINIO

LISTAR conceptos (clases y atributos) candidatos

↓
graficarlos en modelo de dominio

↓
agregar atributos a los conceptos

↓

agregar asociaciones entre los conceptos

- la tarea es identificar las clases conceptuales vinculadas al escenario con el que se trabaja

un modelo de dominio es una representación visual de las clases conceptuales del mundo real en un dominio de interés

- las relaciones entre clases conceptuales deben modelarse con asociaciones

CONTRATOS

- forma de describir el comportamiento en un sistema en detalle
- describen pre y post condiciones

- operación: se detalla el nombre y los parámetros

- pre-condiciones: suposiciones no triviales del estado del modelo antes de la ejecución (se asumen como válidas)

- post-condiciones:
 - cambios en los objetos del dominio
 - declarativas

Resumen Teoría - Objetos 1

HEURÍSTICAS PARA ASIGNACIÓN DE RESPONSABILIDADES (HAR)

→ RESPONSABILIDADES DE LOS OBJETOS

- hacer
- Conocer

→ La asignación de responsabilidades generalmente sucede durante la creación de los diagramas de secuencia

<u>EXPERTO</u>	<u>CREADOR</u>
ALTA COHESIÓN	BAJO ACOPLAMIENTO

EXPERTO → la clase que tiene la info. necesaria para realizar su responsabilidad

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

- B usa objetos A en forma exclusiva
- B contiene objetos A
- B tiene datos para inicializar A

BAJO ACOPLAMIENTO → pocas relaciones con otros objetos (menos dependencia)

ALTA COHESIÓN → clases más fáciles de mantener, entender y reutilizar

TESTING

→ asegurarse que el programa

hace lo que se espera

como se espera

no falla

HERENCIA VS. COMPOSICIÓN

HERENCIA → Superclase

↳ reutilización de CAJA BLANCA → debe conocer todo el código que se hereda

→ débil // acoplados → cambiar algo en la superclase afecta directamente a las subclases/s

→ herencia de estructura vs. herencia de comportamiento

COMPOSICIÓN → variables de instancia que son objetos de otras clases

↳ reutilización de CAJA NEGRA → no se conoce el comportamiento (el código)

→ débil // acoplados → se pueden cambiar más fácilmente los componentes sin afectar el objeto contenedor

SMALL TALK

→ lenguaje OO puro (todo es un objeto; incluso las clases)

→ tipado dinámico //

→ sintaxis minimalista

→ puente de inspiración de todo lo que vino después (en OO)

• Código puente disponible y modificable

• Tiene su propio compilador, debugger, editor, etc.

• Extensible

→ 2 tipos de objetos → los que pueden crear instancias (clases)

↳ los que no

→ Clases son instancias de su metaclasses → una metaclass por cada clase

→ metaclasses instancias de Metaclasses

Resumen Teoría - Objetos I

8

JAVASCRIPT

- dinámico
- basado en objetos (prototipos en lugar de clases)
- multiparadigma
- original / para scripting de pag. web
- fuerte adopción en el lado del servidor (NodeJS)
 - PROTOTIPOS
 - no tengo clases
 - notación estilo JSON
 - cada objeto puede tener su propio comportamiento
 - objetos heredan comportamiento y estado de sus protot.
 - cualquier objeto puede ser prototipo de otro
 - Puedo cambiar el prototipo de un objeto
 - ↳ (y su comportamiento y estado)
 - versión que te hace creer que tienes clases, pero no

↳ compite
con Java

BASES DE DATOS

- orientada a objetos → objetos van a parar a una tabla
- relacional → datos van a parar a tablas
- NoSQL → documentos (como JSON) → menor impedancia
- gráficos
- etc

persistencia
de objetos.