Resumen de Paradigmas de Programación de

Yamila Aylén Soto

Contenido

Conceptos Transversales 5
Paradigmas de Programación 5
¿Qué es un programa? 5
Operación 5
Abstracción 5
Alto nivel y Bajo nivel 5
Expresividad y Declaratividad 6
Expresividad 6
Declaratividad 6
Orden superior 6
Paradigma Lógico 7
Paradigma Estructurado 7
Transparencia Referencial 7
Efecto de Lado 7
Unificación 7
Asignación Destructiva 7
Importancia de los últimos cuatro conceptos 7
Separar la lógica que hace cosas de la que consulta 7
Respetar los contratos blandos 7
Optimizaciones 7
Procesamiento en paralelo 8
Testing 8
Inmutabilidad 8
Declarar vs Instanciar vs Inicializar 8
Estrategias de Evaluación 8
Evaluación Ansiosa/Estricta (Eager Evaluation) 8
Evaluación Perezosa/Diferida (Lazy Evaluation) 8
Deducciones 9
Recursividad 9
Polimorfismo 9
Polimorfismo paramétrico vs ad hoc (Funcional) 9
Repetición de Lógica 9
Esquemas de Tipado 9
Casteo/Casting/Conversión de tipo 9
Sistemas de Tipos 9
Tipado Implícito vs Explícito 10
Chequeo Estático vs Dinámico 10
Detección de Errores 10
Tipos Nominales y Estructurales 10
Algunas combinaciones posibles 10
Clasificaciones 10

Máquina Virtual 10
Unidad n° 1: Paradigma Funcional – Haskell 11
Función 11
Función como caja negra 11
Función como transformación matemática 11
Función desde un punto de vista procedural 11
Función como un TAD 11
Función desde el cálculo lambda 11
Funciones en Haskell 11
Declaración de ~~Variables~~ Etiquetas 11
Declaración de Funciones 11
Inmutabilidad, Efecto y Transparencia Referencial 11
Precedencia de operadores 12
Otros Operadores 12
Pattern matching 12
Variable anónima 12
Errores Comunes 12
Función Parcial 12
Guardas (Funciones Partidas) 12
Errores Comunes 13
Tipado 13
Tipado de Funciones 13
Inferencia 13
¿La inferencia de tipos es una característica del paradigma funcional? 13
Problemas de inferencia en Haskell 13
Variables de Tipo 13
TypeClasses 13
¿Qué tipos pertenecen a cada restricción? 13
La restricción Show 14
DATA 14
Instantiation 14
Acceder a Estructuras/Accessors 14
Azúcar Sintáctico para construir accessors 14
Type Alias 14
Tuplas 14
Pattern Matching Avanzado 14
Patrón Compuesta 15
Manejando el Efecto 15
Currificación 15
Aplicación 15
Aplicación Parcial 15
Notación Point Free 15
Composición 15
Composición de muchos parámetros 16
Orden Superior en Funcional 16
¿La aplicación parcial es también orden superior? 16
Expresiones Lambda 16

Lambdas y Pattern Matching 16
Listas [a] 16
Listas a Bajo Nivel 16
Listas en Alto Nivel 16
Loop Infinito 17
Listas por Compresión 17
Definiciones Locales: Where 17
FOLD / Reducción 17
Listas en No-Tan-Alto Nivel 17
Unidad n° 2: Paradigma Lógico 18
Principio de Universo Cerrado 18
Parámetros 18
Individuos 18
Variables y Variables Incógnitas 18
Predicados 18
Cláusulas 18
Aridad de los Predicados 19
Consultas 19
Conjunción y Disyunción 19
Unificación vs Asignación 19
Predicados simétricos 19
Inversibilidad 19
Generación 20
Casos Posibles de No-Inversibilidad 20
Orden Superior 21
Cuantificador Universal 21
Not vs Forall 21
Functores 21
Polimorfismo en Lógico 22
Listas 22
Predicados de Listas (Alto Nivel) 22
FindAll 22
Is 22
Unidad n° 3: Paradigma Orientado a Objetos – Wollok 23
Objeto 23
Interfaz 23
Estado Interno 23
Identidad 23
Referencias 23
Asignación de Variables 23
Pseudo Variables 24
Inicialización 24
Mensajes y Métodos 24
Tipos de métodos y mensajes 24
Pilares de la Programación Orientada a Objetos 24
Encapsulamiento 24
Delegación y Responsabilidades 25

Polimorfismo 25
Getters y Setters 25
Clases 25
¿Por qué necesito clases? 26
Cosas a recordar 26
Method Lookup 26
Instanciación 26
Herramientas de Instanciación 26
Garbage Collector 26
Igualdad, Identidad, Inmutabilidad 27
Colecciones 27
Sabores de Colecciones en WOLLOK 27
Colecciones y referencias 28
Bloques 28
Mensajes de Colecciones → WOLLOK 28
Con efecto 28
Sin Efecto 29
Herencia 29
Herencia Simple 29
Clase Abstracta 30
Método Abstracto 30
Redefinición 30
Herencia vs Composición 30
Manejo de Errores 31
Lanzado de Excepciones 31
Diagrama de Clases y de Objetos 32
Diagrama estático de Objetos 32
Diagrama de clases 32
Ley de Demeter para Modelar 32

Conceptos Transversales

Concepto>Paradigma>Lenguaje
Idea> Coordinación>Sintaxis

Paradigmas de Programación 

Paradigma: Forma en especial de pensar un problema. Es un conjunto/coordinación de ideas, su influencia se ve principalmente en el momento de modelar una solución a un problema dado.
Paradigma de Programación: Forma en especial de pensar un problema de programación. Usamos *herramientas y conceptos* que cambian en cada paradigma. Un paradigma de programación es un marco conceptual, un conjunto de ideas que describe una forma de entender la construcción de programa, como tal define:

- Las herramientas conceptuales que se pueden utilizar para construir un programa.
- Las formas válidas de combinarlas.

Los distintos lenguajes de programación proveen implementaciones para las herramientas conceptuales descriptas por los paradigmas. Existen lenguajes que se concentran en las ideas de un único paradigma y otros que permiten la combinación de ideas provenientes de varios.
No es suficiente saber en qué lenguaje está construido un programa para saber qué marco conceptual se utilizó en el momento de construirlo. El paradigma tiene más relación con el proceso mental que se realiza para construir un programa que con el programa resultante.

- ✓ Ningún Paradigma es mejor que otro, sino que cada uno presenta ventajas y desventajas frente a un mismo problema y alternativas diferentes para resolverlo.
- ✓ Hay diferentes maneras de clasificarlos y usualmente se utiliza más de una categoría.

¿Qué es un programa?

"Una cosa que usa los recursos de la computadora para resolver un problema"
Un programa es aquello que permite que una computadora realice una tarea determinada.
Cada paradigma ofrece una respuesta distinta a esta pregunta.

Paradigma Estructurado (Estructuras de Datos)

Secuencia ordenada de instrucciones que manipulan un **espacio de memoria**. Hay poca declaratividad→ me enfoco mucho en el cómo. **Mucha imperatividad.**

Paradigma Orientado a Objetos

Conjunto de objetos que se conocen entre sí a través de referencias y se envían mensajes en un ambiente.

Paradigma Lógico

Conjunto de predicados definidos a través de cláusulas (hechos y reglas) que describen propiedades y relaciones de un conjunto de individuos, sobre los que podemos realizar consultas.

Paradigma Funcional

Conjunto de funciones, que pueden ser evaluadas para obtener un resultado.

Funciones: Relaciones que cumplen las propiedades de unicidad y existencia

Operación

Aplicar una función, evaluar un predicado, enviar un mensaje, etc.

Abstracción

Un programa es una entidad muy compleja y por lo tanto es muy difícil abarcarlo en su totalidad. Necesitamos herramientas que nos permitan manejar esa complejidad, las **abstracciones.**
Una abstracción es una forma de interpretar y conceptualizar lo que resulta más importante de una entidad compleja, sin deber tener en cuenta todos sus detalles. Me permite quedarme con lo esencial descartando lo que (para mí, en ese momento) es accesorio. Abstraer es formar una idea general a partir de casos particulares. →Principio de "Dividir y conquistar".
Con abstracciones más adecuadas, responsabilidades repartidas, y sin repetición de lógica, genero un código más expresivo, y más declarativo (al usar orden superior oculto detalles algorítmicos)

Alto nivel y Bajo nivel

Muchas veces se habla de lenguajes de alto y bajo nivel, términos que se refieren al nivel de abstracción de cada lenguaje. Esta clasificación no es un blanco y negro, sino que sirve para comparar entre diferentes lenguajes respecto a qué tan cercanas son sus abstracciones a la máquina (bajo nivel) o al programador (alto nivel).

Expresividad y Declaratividad

Estos dos conceptos son complementarios y vamos a buscar que nuestras soluciones sean lo más declarativas y expresivas que podamos.
No existe EL código expresivo y declarativo, sino que existen códigos más declarativos y expresivos que otros. → Un código puede ser más declarativo y menos expresivo o al revés.

Expresividad

Cuán *entendible* es nuestro código: Cómo nombramos a las variables, funciones, método, etc.
Tiene que ser escrito para que lo entienda cualquier persona, tanto hoy como dentro de unos años.
"El nivel de lindez del código": Escribir un código expresivo es poner atención a las cuestiones que hacen que este código fuente sea más fácil de entender por una persona.
Ejemplo: La solución B es más expresiva.

```
// Solución A
int d(int c[]) {
  int a = 0;
  for (int b = 0; c[b] != NULL; b++) {
    if (c[b] % 2 == 0) {
      a++;
    }
  }
  return a;
}
```

```
// Solución B
int cantidadDeNumerosPares(int* unosNumeros) {
  int cantidadDePares = 0;
  for (int indice = 0;  != NULL; indice++) {
   if (unosNumeros[indice] % 2 == 0) {
      cantidadDePares++;
    }
  }
  return cantidadDePares;
}
```

Motivaciones:

- Que el código sea flexible (que pueda cambiarse fácilmente)
- Que "falle poco" (encontrar y corregir errores tempranamente).
- El desarrollo dura mucho tiempo. (Meses, años)
- El equipo de desarrollo es amplio. (Mucha gente escribiendo el mismo programa).

El código fuente *no puede ser exclusivamente escrito para la computadora*.

¿Cómo lograr la expresividad?

- Usar buenos nombres: Expresar el propósito, ser descriptivo, claro y simple. Está bueno que dé una idea simple de su tipo y que respete las convenciones del lenguaje.
- Usar buenas abstracciones en general (y en particular, la Declaratividad).
- Identar correctamente el código: Separar con espacios. Hace que el código sea legible. No debe abusarse de esto, ni usarlo poco.

Declaratividad

¿Qué quiero que pase?→ No importa tanto el cómo. Pérdida de control (puede ser bueno o malo)
Es una forma de abstracción, que nos permite describir el conocimiento relativo a un problema desentendiéndonos de los algoritmos necesarios para manipular esa lógica, que son provistos por el motor. En un programa construido de forma declarativa se produce una separación entre la descripción del problema por un lado y los algoritmos o estrategias para encontrar la solución por el otro. Un error común es hablar de la separación entre el qué y el cómo. Un programa declarativo separa claramente los siguientes elementos:

- El objetivo
- El conocimiento
- El motor que manipula el conocimiento para lograr el objetivo deseado

Ejemplo: C es más declarativa (esNumeroPar)

```
// Solución B
int cantidadDeNumerosPares(int* unosNumeros) {
  int cantidadDePares = 0;
  for (int indice = 0;  != NULL; indice++) {
    if (unosNumeros[indice] % 2 == 0) {
      cantidadDePares++;
    }  }
  return cantidadDePares;
}
```

```
// Solución C
int cantidadDeNumerosPares(int* unosNumeros) {
  int cantidadDePares = 0;
  for (int indice = 0;  != NULL; indice++) {
    if (esNumeroPar(unosNumeros[indice])) {
      cantidadDePares++;
    }  }
  return cantidadDePares;
}
```

Orden superior

Una operación es de orden superior si la misma recibe otra operación (comportamiento) por parámetro, siendo capaz de ejecutarla internamente. Ventajas de usarlo:

- Puedo aislar y reutilizar comportamiento común.
- Puedo partir mi problema, separando responsabilidades, entre el código que tiene orden superior, y el comportamiento parametrizado.
- Puedo tener un código con partes incompletas, esperando rellenarlos pasando comportamiento por parámetro, y no sólo datos.
- ¡Puedo generar mejores abstracciones! (En la materia lo vemos en Haskell más directamente, aunque se puede también en SWI-Prolog).

Paradigma Lógico

Decimos que un predicado es de Orden Superior si este recibe como argumento una consulta a otro/s predicado/s. Algunos ejemplos son not, findall y forall.

Paradigma Estructurado

Una porción de código puede alcanzar resultados algorítmicos como si fuesen obtenidos a través de funciones de orden superior, ejecutando código dinámicamente durante la evaluación.

¿Y los objetos?

Un X que puede admitir otro X como parametro. Digo que reconozco dos órdenes, si eso es una característica especial que quiero remarcar, si no... lo que pasa es que no tiene sentido la distinción de órdenes en este contexto para cada paradigma: si tiene sentido decir algo así como “orden superior”, y en qué casos para qué uso la idea superior, en particular en funcional.

Transparencia Referencial

Se tiene si en el programa podemos reemplazar todas las operaciones por su resultado y se obtiene el mismo efecto.

Propiedades necesarias para tener Transparencia Referencial.

- Independiente: No dependen del estado de nada que este fuera de sí misma
- Sin estado/Stateless: No tiene un estado que se mantenga de llamada en llamada
- Determinística: Siempre devuelven el mismo valor dados los mismos argumentos
- No produce efecto colateral

Efecto de Lado

Hay efecto cuando un cambio de estado sobrevive a la realización de una operación. Cambios que sobreviven el contexto que se evaluó (guardar en disco, pisar una variable, imprimir una pantalla). Cualquier operación que esperamos que cause algo que viva luego de que la operación terminó. No hay efecto cuando “Meto la mano en la licuadora y la saco intacta” Otra definición válida es: Si le sacás una foto al sistema (llamémosla F1), después realizas la operación de tu interés, y le volvés a sacar una foto al sistema (F2). Si F1 y F2 son distintas => la operación tiene efecto de lado.

Unificación

Unificar es encontrar una sustitución capaz de igualar 2 términos. Cuando se efectiviza está sustitución hablamos de ligado de variables (tal valor se ligó a tal variable).

Asignación Destructiva

Asignar destructivamente es reemplazar el valor de una variable por otro valor.
La unificación no se considera asignación, al momento de ligar no había ningún valor anterior.

Importancia de los últimos cuatro conceptos

Separar la lógica que hace cosas de la que consulta

Muy seguido vemos operaciones que tienen efecto y a su vez retornan algún valor relacionado con el mismo, estas prácticas pueden llevar a confusiones que producen un funcionamiento erróneo.

Respetar los contratos blandos

Un contrato blando es algo que cierta pieza de código requiere que cumpla el usuario para que la misma funcione de la forma esperada, pero esos requisitos no son validados de ninguna forma.

Optimizaciones

Tener asegurada la transparencia referencial permite hacer optimizaciones como las que tiene el

motor de Haskell que afectan globalmente a los programas construidos con el mismo. La evaluación perezosa o lazy es posible gracias a esta característica. También lo podemos ver en Prolog que para buscar soluciones utiliza el mecanismo de Backtracking de modo que se puedan encontrar múltiples respuestas a una consulta, así como descartar los caminos por los cuales no sea posible hallar alguna, de una forma eficiente.

Procesamiento en paralelo

Si aseguro que evaluar el criterio de filtrado sobre cada elemento no va a provocar ningún efecto que pueda alterar mi resultado final, podría permitir aprovechar mejor el hardware disponible.

Testing

El testeo unitario se basa en la premisa de que cada test sea independiente del otro y eso se logra controlando que el estado del sistema antes y después de correr cada test sea el mismo, por ese motivo es importante mantener el efecto controlado y poder revertir aquellos cambios que sobrevivan a la ejecución de cada test particular.

Inmutabilidad

Está asociado a la **ausencia de cambio**. En los paradigmas Funcional y Lógico, la inmutabilidad está garantizada, ya que no es posible modificar los datos con los que trabaja una función o un predicado, en todo caso lo que se puede hacer para emular un cambio sobre una estructura es retornar o relacionar con una nueva estructura a partir de la original con la información que tendría como consecuencia de la transformación deseada. En cambio, en los paradigmas con efecto colateral, esto es una decisión de diseño: En el paradigma Orientado a Objetos, hay objetos inmutables, pero no son todos los casos: Decimos que **un objeto es inmutable si no puede cambiar su estado interno** (su conjunto de atributos) después de su inicialización. Si la interfaz del objeto tiene una forma de inicializar sus variables, pero no exhibe el comportamiento para settear sus atributos, sus usuarios no podrán alterar su estado interno más adelante.

Declarar vs Instanciar vs Inicializar

Declarar: Es una construcción de lenguaje que especifica las propiedades de un identificador, declara lo que una palabra (identificador) "significa". Crear las variables antes de poder usarlas en nuestros programas, indicando qué nombre va a tener y qué tipo de información va a almacenar,
Instanciar: Llamamos instanciación a la creación de un objeto a partir de una clase, la cual define cuál es el comportamiento y los atributos que debería tener dicha instancia.
Inicializar: Es la asignación de un valor inicial para un objeto de datos o una variable.

Estrategias de Evaluación

Manera en cómo un programa o una herramienta/motor de reducción, trabaja sobre una expresión para llegar a un resultado. A una expresión que consta de una función aplicada a uno o más parámetros y que puede ser “reducida” aplicando dicha función la vamos a llamar Redex (Reducible Expression). Se le dice reducción al hecho de aplicar la función.

Evaluación Ansiosa/Estricta (Eager Evaluation)

Trabaja de adentro hacia afuera: Opera las funciones que no son principales para después operar con la principal. Ejemplo: cuadrado (siguiente 3) → cuadrado (3 +1) → cuadrado 4 → 4*4 →16

Call by value/ Innermost evaluation

“Voy de adentro hacia afuera”. Elige el redex que está “más adentro” entendiendo por esto al redex que no contiene otro redex. Si existe más de un redex que cumple dicha condición se elige el que está más a la izquierda. Esta estrategia me asegura que los parámetros de una función están completamente evaluados antes de que la función sea aplicada, “se pasan por valor”.

Evaluación Perezosa/Diferida (Lazy Evaluation)

Trabaja desde afuera hacia adentro: Opera las funciones principales primero y luego reduce las que se pasan por parámetro. Ejemplo: siguiente 3 * siguiente 3 → (3+1) * (3+1)→ 4*4 → 16
Para que una condición devuelva valores finitos o infinitos, debe evaluar algo finito. (HASKELL)

Lazy Evaluation = call by name + sharing

Call by name / Outtermost evaluation

Llamo a la operación y después evalúo las expresiones que toma por parámetro. No evalúo los parámetros sino hasta que los necesito. Elige el redex que está “más afuera” entendiendo por esto

al redex que no está contenido en otro redex. Si existe más de un redex que cumple dicha condición se elige el que está más a la izquierda. Las funciones se aplican antes que los parámetros sean evaluados, "Se pasan por nombre".
Nota: Hay que tener en cuenta que muchas operaciones requieren que sus parámetros estén evaluados antes de que la función sea aplicada, por ejemplo: "+". A las funciones que cumplen con esta propiedad las vamos a llamar funciones estrictas: Aritméticas y Pattern Matching.

Deducciones

Cualquier evaluación de una expresión que llega a un resultado, llega al mismo resultado siempre. Si existe un resultado para una expresión, una evaluación perezosa va a encontrarlo, mientras que una ansiosa no podría hacerlo.
Ejemplo: Con lazy Evaluation: const x _ = x & loop = loop
>const 42 loop → 42 no necesito evaluar el segundo parámetro

Recursividad

Una operación es recursiva si en su definición se invoca a sí misma. Debe tener un caso base.

Polimorfismo

Capacidad de una operación de recibir por parámetro valores de distinto tipo. Poder tratar dos cosas diferentes de forma diferente, sin preguntar qué forma tienen.

Polimorfismo paramétrico vs ad hoc (Funcional)

Cuando hablamos de polimorfismo paramétrico tenemos una sola definición de la función, en cambio cuando tenemos polimorfismo ad-hoc tenemos muchas definiciones para la misma función de modo que se puedan soportar distintos tipos.
Típicas funciones con polimorfismo paramétrico son las que operan sobre listas en Haskell: filter, length, foldl, etc. No necesariamente pueden recibir "cualquier valor", depende de lo que hagas.

Repetición de Lógica

Sucede cuando 2 o más operaciones diferentes hacen los mismos pasos en toda o parte de su aplicación. Es necesario abstraer esa parte y crear una operación auxiliar a la que invocaremos cuando sea necesario. Es importante ya que cualquier cambio que se produzca en una parte del código, va a tener que cambiarse en todas las partes donde esa lógica se repita. ¡Está mal!

Esquemas de Tipado

Tipo: Toda expresión en un programa puede denotar diferentes valores en diferentes momentos. Un tipo describe un conjunto de valores. Al asignarle un tipo a una expresión se está delimitando cuál es el conjunto de valores que podría denotar. Las operaciones computacionales en general no pueden ser aplicadas sobre cualquier valor, un sistema de tipos me provee una forma de estudiar qué valores tienen sentido para ser utilizados en una. La idea de tipo nos permite relacionar:

- Un conjunto de valores que tienen ese tipo o son de ese tipo,
- Con las operaciones que pueden ser realizadas sobre esos valores.

Casteo/Casting/Conversión de tipo

Es un procedimiento para transformar una variable primitiva de un tipo a otro, o transformar un objeto de una clase a otra clase siempre y cuando haya una relación de herencia entre ambas

Sistemas de Tipos

(Existen diferentes tendencias en cuanto a cómo se clasifican a los diferentes lenguajes en función de la presencia o no de un sistema de tipos o de las características de ese sistema de tipos).
lenguaje tipado es el que asocia cada expresión con un tipo (con uno no trivial). El sistema de tipos es el componente que administra la información de tipos de un programa. Un lenguaje se considera tipado por la existencia de un sistema de tipos, independientemente de que la sintaxis del lenguaje incorpore información de tipos. Los objetivos de un sistema de tipos son:

- Ayudar a detectar errores al programar.
- Guiar al programador sobre las operaciones válidas en un determinado contexto, tanto en cuanto a documentación como en cuanto a ayudas automáticas que puede proveer.
- En algunos casos el comportamiento de una operación puede variar en función del tipo de los elementos involucrados en la misma.

Algunas características de un sistema de tipos:

- Proveen información al programador, de forma más precisa que un comentario.
- Debería permitir hacer validaciones utilizando la información de tipos, algorítmicamente.
- Las validaciones basadas en un sistema de tipos son más fácilmente automatizables que otros tipos de especificaciones formales.

Tipado Implícito vs Explícito

Explícito: Toda variable, parámetro, método tiene un tipo definido. Para que dos objetos puedan ser polimórficos tengo que indicarlo explícitamente: herencia o interfaces. Ejemplo: Java
Implícito: No necesito hacer una indicación explícita de que dos objetos sean polimórficos, basta con que entiendan algún(os) mensaje(s) en común. Ejemplo: Smalltalk

Chequeo Estático vs Dinámico

En función de en qué momento se chequean: Si tengo un error de tipos, cuando tenemos chequeo estático, el programa no compila. En cambio, en el chequeo dinámico rompe en tiempos de ejecución. Pero, ante la presencia de casteos el chequeo estático se pierde.
Finalmente es importante diferenciar entre chequeo dinámico y no chequeo. El chequeo dinámico me permite manejar el problema y manejarlo o por lo menos me lo informa correctamente, la ausencia de chequeo suele causar errores muy difíciles de detectar o corregir.

Detección de Errores

Tipos de Errores

Trapped (atrapados): Son los que se detectan inmediatamente, por ejem. una división por cero.
Untrapped (no atrapados): Son errores que pueden no ser detectados. El programa podría continuar ejecutándose por un tiempo antes de detectar el problema. Esto puede ocurrir en algunos lenguajes por ejemplo si se accede a posiciones de un array más allá de su longitud.

Tipos de Programas y Lenguajes

Un programa se considera seguro si no causa Untrapped errors. Un lenguaje se considera seguro si todo programa escrito en ese lenguaje no tiene de Untrapped errors.
En todo lenguaje se puede designar un conjunto de los errores como prohibidos. Los errores prohibidos deberían incluir a todos los errores Untrapped, más un subconjunto de los errores trapped. Un programa que no causa errores prohibidos se dice que tiene buen comportamiento.
En un lenguaje strongly checked, todos los programas tienen buen comportamiento, es decir:

- No ocurren errores Untrapped.
- No ocurren los errores trapped que se consideran prohibidos.

Queda a responsabilidad del programador evitar los errores trapped que no sean prohibidos.

Tipos Nominales y Estructurales

Llamamos nominales a los tipos que tienen un nombre.

Algunas combinaciones posibles

Suele haber una fuerte relación entre los tipos nominales-explícitos y estructurales-implícitos. Esto es porque uno para explicitarlos le pone nombre. También suele haber una fuerte relación entre el chequeo estático y el tipado explícito o nominal.

Clasificaciones

En resumen planteamos tres clasificaciones:

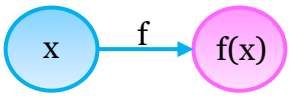
- En cuanto a la forma de chequeo puede ser estático/compile time, dinámico/runtime o nada.
- En cuanto a la forma de especificar el tipo de algo puede ser explícito o implícito / inferido.
- En cuanto a la forma de constituir un tipo puede ser nominal o estructural (no sé si agregar dependientes, también hay más variantes).

Máquina Virtual

Software que simula un sistema de computación y puede ejecutar programas como si fuese una computadora real →"un duplicado eficiente y aislado de una máquina física".

Unidad n° 1: Paradigma Funcional – Haskell

Función



Según este paradigma, cada operación es una FUNCIÓN: Relación entre dominio e imagen. Para una entrada tenemos una salida (existencia) que es única (unicidad). Veo los problemas como preguntas y las soluciones como resultados de estas.

Función como caja negra

Una forma simple de pensar una función es como una máquina con una salida y al menos una entrada, capaz de producir un resultado.

Función como transformación matemática

Las funciones son transformaciones matemáticas que presentan transparencia referencial.

Función desde un punto de vista procedural

No tienen efecto, su aplicación no afecta al contexto, o, cuando menos, no es visible para el observador que evalúa la expresión. Las funciones no pueden mutar sus argumentos ni otras variables. Esto se garantiza al eliminar la asignación destructiva del lenguaje.

Función como un TAD

Su única operación primitiva es la aplicación, definida entre una función y otro valor. Esta operación, a su vez también es una función, llamada apply, (($) en el Prelude de Haskell). Las demás operaciones complementarias, como la composición, se construyen a partir de la aplicación.

Función desde el cálculo lambda

Desde el punto de vista del cálculo lambda, la función es LA primitiva del lenguaje, y todas las funciones son anónimas, es decir, son expresiones lambda.

Funciones en Haskell

Las funciones en Haskell presentan todas las características mencionadas anteriormente.

- Son transformaciones matemáticas, que presentan transparencia referencial: libres de efecto
- Las funciones son valores
- Las funciones tienen tipo función ((->) a b), que está determinado por su dominio e imagen.
- Son un caso particular de las relaciones, presentan unicidad y existencia para todo su dominio.
- Están currificadas, no existen funciones de más de un argumento, sino que se emulan a partir de funciones de un argumento que devuelven otra función que toma los parámetros restantes.
- El mecanismo de la evaluación de las funciones es la reducción (reducción **β**)
- La única operación primitiva del tipo función es la aplicación, por la cual se evalúa una función pasándole sus argumentos y obteniendo un resultado, sólo es función aquello y sólo lo que pueda ser aplicado. **Moraleja:** no tiene sentido hablar de funciones de cero argumentos.

Declaración de ~~Variables~~ Etiquetas

No son espacios en memoria, sino que son incógnitas temporales que se resuelven cuando quiero conocer el valor de una función en ese punto del Dominio. → Le ponemos un “alias” o “etiqueta” a un valor. → nombreAliasEtiqueta = Valor

Declaración de Funciones

Las funciones van a ser nuestras herramientas para poder operar a los valores. Definir Funciones: nombre P1 P2 P3 ... = Función → El ‘=’ funciona para decir que dos expresiones son equivalentes

1) Nombre: Alias/etiqueta que adopta la función
2) Parámetros: Pueden ser tantos como se necesiten y se separan por espacios.
3) Función: Es lo que se debe realizar. Puede devolver todo tipo de valor.
    a. Infija: Los operadores se ubican como si fueran una operación matemática: = P1+P2
    b. Prefija: Los operadores se ponen entre paréntesis inmediatamente después del = y los parámetros se separan con espacios: = (+) P1 P2

Tanto las variables como las funciones comienzan con minúscula. → Los tipos con mayúscula

Inmutabilidad, Efecto y Transparencia Referencial

En Haskell **no hay efecto**. Esto quiere decir que los valores igualados no van a mutar luego de ser operados por las funciones. Este concepto se llama **inmutabilidad**. Cuando *declaro* una variable, función, etc. con un valor, el mismo no cambiará nunca. Por esto, es que en Haskell logramos tener lo que se llama **transparencia referencial**.

```
> frecuenciaCardiaca
=> 80
> correr (frecuenciaCardiaca)
=> 130
> frecuenciaCardiaca
=> 80
```

Precedencia de operadores

Precedencia	"Operador"	(De mayor a menor precedencia)
11	(,)	
10	Aplicación Normal	(Existe la parcial)
9	.	Composición
8	^, ↑	Elevación
7	*,/	
6	+,-	
5	:	Separa el primer elemento en una lista (head : tail)
4	==, /=, <, <=, >, >=	Comparar valores del tipo Ord
3	&&	
2	||	
1	$	Se usa para la composición

Otros Operadores

mod , ++, !!

```
gustoFavorito "Yamila" = "Sabayón"
gustoFavorito "Axel" = "Limón"
gustoFavorito n = "Vainilla"
    ➔ Los nombres forman PATRONES
```

Pattern matching

Defino una función en múltiples líneas (varias sentencias). Puedo declarar sentencias que no contengan variables, sino **patrones** (valores que puede tomar un parámetro). Se ejecuta una sentencia si el valor del parámetro coincide con la del patrón.

IMPORTANTE: El programa toma la primer opción que cumpla la condición, si no la encuentra, tira error. Hay que tener cuidado en el orden en que ponemos las condiciones (Primero patrones).

Ventaja: Simplifica mucho la codificación, ya que sólo escribimos la forma de lo que esperamos y podemos desglosar los componentes de estructuras complejas.
Desventaja: Para acceder a los elementos de las tuplas, nuestro código se verá muy afectado ante un cambio en las estructuras manejadas.

Variable anónima

```
gustoFavorito "Yamila" = "Sabayón"
gustoFavorito _ = "Vainilla"
    ➔ Los nombres forman PATRONES
```

Machea con cualquier valor del patrón, es nuestra forma de decir "Otro caso". Por lo dicho anteriormente, debe incorporarse al final. Al ser un Patrón, siempre está a la izquierda del 'igual'. Si quiero usarlas a la derecha del igual (que sean parte de lo que devuelvo) tengo que usar variables comunes.

```
:: TIPO
Patrón = Valor
```

Errores Comunes

- Repetir lógica.
- No tiene sentido representar funciones que devuelven "true o false".
- No poner la variable anónima o todos los casos posibles. No contamos con el concepto de "Universo Cerrado", por lo que si no se pone esta "salida", va a romper.

Función Parcial

No está definida para todos los posibles valores de los parámetros. Es difícil que una función ande con cualquier dominio. A veces puede pasar que una función completamente válida entre en un "loop" infinito sólo por consultarla mal. → Hay formas de "Restringir" ciertas cosas, pero quien haga la consulta puede introducir cualquier valor, y romper el programa de todas formas.

Guardas (Funciones Partidas)

Las funciones partidas o definidas por partes o por trozos son funciones que para diferentes valores del dominio, tienen una definición diferente. Por ejemplo, la función Módulo
En Haskell, las funciones partidas se escriben con Guardas, y se escribe la imagen de cada parte a la derecha del igual:

```
f x   | x >= 0 = x
      | x <0  = -x
```

➔ Cada | es una opción nueva.

➔ El programa toma (como siempre) la primer opción que cumpla las condiciones.
➔ Otherwise → Default. Se usa AL FINAL, para decir que es "Otro caso".

Errores Comunes

- No debemos confundir el uso de guardas con las funciones que devuelven booleanos, esto es un uso incorrecto de las guardas. En otras palabras: Si una expresión es verdadera o falsa por sí sola, no se lo debo preguntar. (Las condiciones son booleanos, no los resultados)
- Las funciones por partes no son la única forma de tener diferentes definiciones para una función, existen casos en los cuales alcanza con el uso de Pattern matching (y es más declarativo).
- Hacer la guarda antes de tiempo
- Repetir lógica.
- No poner Otherwise o todos los casos posibles. No contamos con un "Universo Cerrado".

```
gustoFavorito nombre
  | nombre == "Yamila" = "Sabayón"
  | nombre == "Axel" = "Limón"
  | otherwise = "Vainilla"
Incorrecto: Repetición de Lógica.
```

Tipado

Tiene muchas ventajas, nos permite detectar errores, restringir el dominio

Tipado de Funciones

nombreFuncion :: TipoP1 → TipoP2 → .... → TipoResultado
Se ponen tantas flechas como parámetros le "llegan" a la función.

>:t nombreFuncion
Nos devuelve el tipo

Inferencia

Es la capacidad que tienen algunos lenguajes con chequeo estático de tipos para calcular el tipo de una expresión, función o variable sin necesidad de contar con anotaciones de tipo.

¿La inferencia de tipos es una característica del paradigma funcional?

Es un concepto que fue desarrollado primero en el contexto del paradigma funcional, pero no está limitada este tipo de lenguajes y cada vez más está siendo utilizada en todo tipo de lenguajes. La inferencia es más simple en lenguajes que no permiten realizar asignaciones. Haskell es capaz de inferir tipos, es decir, no hace falta que lo aclaremos siempre. Podemos aclarar el tipo de una función siempre y cuando eso tenga sentido.

Problemas de inferencia en Haskell

Haskell puede inferir correctamente el tipo de las funciones que definimos casi siempre, pero existen casos en los cuales necesita un poco de ayuda de parte del programador: Va a tratar de asignarle el tipo más genérico posible a las cosas, para que la función sea lo más útil posible. Por ello nosotros podemos inferir un tipo más específico si queremos.

Variables de Tipo

Son variables que toman formas diferentes dependiendo de con qué parámetro la ejecuto. Admite cualquier cosa, es una especie de comodín. Ejemplo:

```
ignorarElPrimero x y = y
ignorarElPrimero :: a → b → b
```

TypeClasses

Son un contrato o tipo de datos abstracto que agrupa las funciones sobrecargadas: definen una lista de funciones que un tipo deber implementar para considerarse de esa clase. No es un tipo concreto, sino una restricción sobre una variable de tipo (que puede tomar como valores posibles tipos concretos). Se anotan con una Variable de Tipo. Ejemplo: doble :: Num a => a -> a

¿Qué tipos pertenecen a cada restricción?

- Eq: Todo aquello que puedo comparar con (==) y (/=). Para las listas y tuplas, son equiparables si los tipos que las componen lo son. Las funciones no son equiparables.
- Ord: Todo aquello que puedo comparar con (==), (/=), (<), (>), (<=) y (>=). Además de incluir a la mayoría de los tipos numéricos, incluye a los caracteres y los booleanos (hay un orden preestablecido para los booleanos). También incluye a las listas y las tuplas siempre y cuando los tipos que las compongan sean ordenables. Las funciones no son ordenables.
- Num: Los tipos que integran esta familia son Complex, Int y Float. No todos los Num son Ord.
- ¿Y los data? Algo muy común es que queramos que nuestros tipos de datos sean equiparables. Si **los tipos de datos que componen a nuestros data lo son**, alcanza con derivar Eq y no

necesitaremos definir la igualdad. De lo contrario, para que sea Eq será necesario declarar que nuestro tipo de dato es instancia de la typeclass Eq e incluir una definición para la función (==).

La restricción Show

Los resultados de nuestras funciones son valores que para mostrarse por pantalla, deben tener una representación en forma de cadena de caracteres. La magia la realiza la función **show:**: (Show a) => a -> String

No todos los valores pueden ser parámetro de la función show, las funciones no tienen una representación en String. La única función que se define en la restricción Show es la función show.

¿Qué tipos pertenecen a la restricción Show?

Bool, Char, Double, Float, Int, Integer, (Show a) => [a] –Listas, (Show a,b) => (a,b) –Tuplas

Resumiendo: casi todos los tipos menos función.

deriving (Show, Eq)
Sirve para que los data se puedan mostrar por consola y comparar por igualdad

DATA

Lo usamos para definir un nuevo tipo de dato

data <NombreDelTipo> = <Constructor> <TipoDeLosCampos> deriving (Show, Eq)

Constructor: Función con la que voy a poder instanciar a estos tipos que estoy describiendo.

Ejemplo: data Alumno = Alu String String Int deriving (Show, Eq)
→Alu :: String -> String -> Int -> Alumno

Instantiation

Se pone el nombre del nuevo parámetro, y se lo iguala al constructor y los campos.

alumno1 :: Alumno
alumno1 = Alu nombreAlumno1 legajoAluno1 dniAlumno1 → alumno1 :: Alumno

También se pueden escribir los parámetros por Extensión: data Bool = True | False

Acceder a Estructuras/Accessors

Necesitamos Funciones Auxiliares, que tomen al constructor y los tipos de los campos entre paréntesis y devuelvan el campo que queramos. (Es un Pattern Matching)

Ejemplo: nota (Alu _ _ nota) = nota Consulta: > nota alumno1

Azúcar Sintáctico para construir accessors

```
data Alumno = Alu {          alumno1 = Alu {
      nombre :: String,        nombre = "José",
      legajo :: String,        legajo = "166-864.8",
      nota:: Int               nota = 8
} deriving (Show, Eq)        }
```

alumno1 = Alu "José" "166-864.8" 8

Type Alias

Es otra forma de llamar a los tipos. Se usan para dar más contexto a los tipos que uso, nos ayuda a entender más qué es lo que esperamos de una función. ¡No construye otro tipo!

Ejemplo: type Nombre = String → Alu :: Nombre -> String -> Int -> Alumno

Tuplas

(Casi siempre que vemos tuplas, podemos usar Datas). Es un "data anónimo", no tiene nombre. Pongo entre paréntesis los datos que quiera, separados por comas, y eso me construye una tupla. Tienen una "Aridad", que es la cantidad de datos/parámetros que les paso. Cada parámetro tiene un tipo, es por ello por lo que el tipo de la tupa son los tipos de los parámetros, entre paréntesis y separados por comas. Ejemplo: ("Yamila Soto", "167-222.8", 10) → : t (String, String, Int)

Se usan para modelar datos muy genéricos, para tuplas con el mismo tipo, no puedo definir si tienen un contexto diferente. Es decir, como modelé el ejemplo anterior, también puedo modelar el nombre de un perro, su raza y edad: ("Mindy", "Boxer", 4)

Accessors: También se crean con Pattern Matching. Ejemplo: raza (_, unaRaza, _) = unaRaza.

Pattern Matching Avanzado

```
data Nota = Nota{          id :: Nota -> Nota
  valor :: Int,            id (Nota elValor elDetalle) = Nota elValor elDetalle
  detalle :: String        id nota@(Nota elValor elDetalle) = nota
} deriving (Show, Eq)        →@: herramienta que me sirve para darle nombre a todo un patrón.
```

Patrón Compuesta

Forma de accesar a un data, dentro de otro data.

```
data Alumno = Alumno{
legajo :: String,
notaFuncional :: Nota,
notaLogico :: Nota,
notaObjetos :: Nota
} deriving (Eq, Show)
```

```
notaFinal (Alumno _ (Nota notaFuncional _) (Nota notaLogico _)
(Nota notaObjetos) ) =
(notaFuncional + notaLogico + notaObjetos) /3
```

→ En realidad, lo mejor es utilizar accessors (los tengo que crear):

```
notaFinal alumno = valor (notaFuncional alumno) + valor
(notaLogico alumno) + valor (notaObjetos alumno)
```

Manejando el Efecto

Lo representamos modelando "cómo sería el mundo si el cambio se produjera", no modifico nada realmente. Recibo un parámetro y retorno ese parámetro emulando cómo sería si se aplicara la modificación. Eso es lo que utilizo después. Construyo una estructura igual con las modificaciones. Ejemplo: Incrementar la nota de un alum →subirNota unaNota = Nota {valor= valor unaNota +1}

Currificación

Técnica que consiste en transformar una función que recibe múltiples variables, en una secuencia de funciones que utilizan un único argumento. → Todas las funciones están currificadas.
f :: a -> b -> c -> d → es una función que recibe un parámetro de tipo a y devuelve una función de tipo (b -> c -> d). Esa función, a su vez, espera un parámetro de tipo b y devuelve una función de tipo (c -> d). Finalmente, esta es una función que espera un c y devuelve un d: f x y z == ((f x) y) z

Aplicación

Cuando se aplica una función a unos argumentos, el resultado se obtiene sustituyendo esos argumentos en el cuerpo de la función respetando el nombre que se les dio a cada uno. Es posible que esto produzca un resultado que no puede ser reducido; pero es más común que el resultado de esa sustitución sea otra expresión que contenga otra aplicación de funciones, entonces se vuelve a hacer el mismo proceso hasta producir un resultado que no pueda ser reducido (resultado final).

Aplicación Parcial

Sucede cuando paso menos parámetros de lo esperado→ Se aplican solo una porción de los parámetros. Todas las funciones de Haskell pueden ser aplicadas parcialmente porque todas están currificadas. "Se guarda el/los parámetros que le pasé, hasta que ponga los que faltan". Ejemplo:

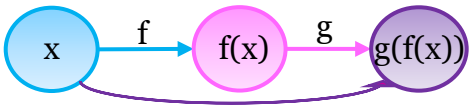
```
> doble n = 2 * n          > doble n = (2*) n          > doble = (2*)
```

Se dice **que siempre sucede después del igual**.

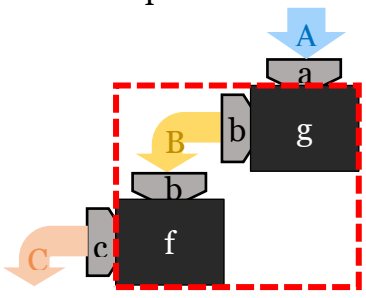
Notación Point Free

Puedo eliminar la aplicación de un parámetro cuando se usa de un lado y del otro, a la derecha de todo. No cambia el tipo de la función ni su funcionamiento. No es posible en todas las funciones.
promociona alumno = (not.(<=8).nota) alumno → promociona = not. (<=8).nota

Composición



También es válida la Aplicación anidada (más difícil de leer).
(.) :: (b->c) -> (a -> b) -> (a -> c)
Lo mismo que con la composición de funciones matemáticas → FoG(x). Primero se aplica la función de la derecha con el valor y luego se aplica la de la izquierda con el valor que nos devolvió la función anterior. Como en matemática, el valor que retorne la función de la derecha tiene que ser un valor que la función de la izquierda pueda operar: La imagen de la función de la derecha esté incluida en el dominio de la función de la izquierda. Se denota con un punto. Si quisiéramos componerlo al revés: Va a romper 💥
Porque espera un tipo diferente. Es por esto mismo que necesitamos el punto, porque si no lo ponemos, estaríamos pasando una función como parámetro y podría romper.



Lo que nos queda después de componer dos funciones es... ¡una nueva función! 😮
Como no podemos pisar valores con variables, la composición nos permite encadenar las funciones para trabajar con diferentes valores y así poder crear soluciones más complejas. ✨

```
> cuádruple n = (doble.doble) n    > cuádruple n = doble.doble $ n    > cuádruple = doble.doble
```

Si pongo un "." es composición, sino es aplicación. El operador (.) tiene poca precedencia.

Composición de muchos parámetros

Usamos aplicación parcial: La uso sobre funciones que esperan n parámetros, para convertirlas en funciones que esperan 1 solo. Pongo los parámetros que se ubican "más a la izquierda" para que reciba otro. También puedo usar la función **flip**, que cambia de lugar los parámetros.

Orden Superior en Funcional

Se definen las funciones de orden superior como funciones que **reciben funciones por parámetro** o bien **devuelven una función como resultado**. Algunas funciones comunes son filter, map, fold, funciones de ordenamiento o búsqueda, composición de funciones (.), flip, etc. Ejemplos comunes del análisis matemático son la derivada y la integral.

Ejemplo: ganaSegun criterio carta1 carta2 = criterio carta1 > criterio carta2
ganaSegunVelocidad = ganaCarta1Segun velocidad

¿La aplicación parcial es también orden superior?

La Aplicación Parcial no es una función, lo que sí podemos analizar es la Aplicación y lo importante es qué recibe y qué devuelve, eso se ve cuando se analiza el tipo de la función. Existe una función que nos va a servir para realizar este análisis llamada $ que lo que hace es aplicarle un parámetro a una función.

```
($) :: (a -> b) -> a -> b
($) funcion parametro = funcion parametro
```

Expresiones Lambda

Herramienta que sirve para crear funciones "anónimas": no tienen nombre, se usan y se descartan. Las uso usualmente cuando necesito delegar una función pero sé que nunca más la usaré, y no tiene sentido crearla. Define la función en el momento y la pasa como un parámetro. Se aclara los parámetros que recibe y lo que retorna. Sintaxis: (\ <parámetros> -> <expresión>)
Ejemplo: ganadorSegunIMC = ganadorSegun (\carta -> peso carta div altura carta ^2)
Si no le damos un nombre a nuestras funciones podríamos perder abstracciones útiles que podrían luego ser utilizadas en otros puntos, es importante ser criteriosos respecto su uso.

Lambdas y Pattern Matching

Algo interesante que se puede hacer con las expresiones lambda es descomponer sus parámetros usando Pattern matching como cuando definimos funciones normales.

Listas [a]

Siempre la lista es de un tipo. No puedo mezclar distintos tipos en las listas.

Tiene dos constructores: $[a] = \begin{cases} [\,] -- Nil :: [a], & es\ la\ lista\ vacía\ de\ tipo\ a. Caso\ semilla \\ (:) -- Cons :: a \to [a] \to [\,], & recibe\ un\ elemento\ y\ una\ lista \\ & cabeza \to cola \end{cases}$

Son estructuras recursivas (compuestas por una cabeza y una cola que a su vez es una lista) la forma más natural para trabajar con ellas es recursivamente. La lista es una estructura vacía o una que tiene una cabeza y una cola. Si la lista no está vacía, podemos dividirla en cabeza y cola tantas veces como sea necesario, hasta tener una lista vacía. Nota: String == [Char]

Ejemplo: [1,2,3] == 1 : 2 : 3 : [ ] :: [Int] → 1 : [2, 3] == 1 : 2 : [3] == 1 : 2 : 3 : [3]

Listas a Bajo Nivel

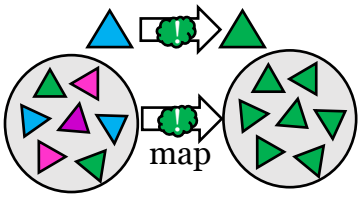
```
head :: [a] -> a
head (x : _) = x

tail :: [a] -> [a]
tail (x : xs) = xs

length :: [a] ->Int
length [] = 0
length (_: xs) = 1 + length xs

elem :: Eq a => a -> [a] -> Bool
elem _ [] = False
elem x (y : ys) = y ==x || elem x ys

null :: [a] -> Bool
null [ ] = True
null _ = False

estaOrdenada :: Ord a => [a] -> Bool
estaOrdenada [] =True
estaOrdenada [x] = True
estaOrdenada (x:y:z) = x<y && estaOrdenada z
```

Listas en Alto Nivel

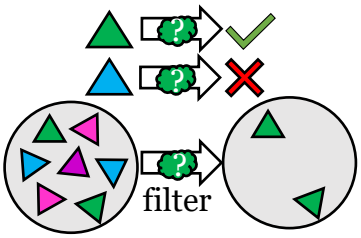
Producimos resultados a partir de cada elemento de la lista

Función map

map :: (a -> b) -> [a] -> [b]



Es una función que recibe una lista y a cada uno de los elementos de esta, le aplica una función (condición), que también le pasamos por parámetro.

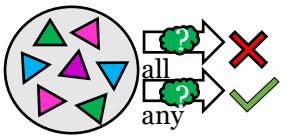


Función Filter

filter :: (a -> Bool) -> [a] -> [b]

A Partir de un conjunto, me quedo con los elementos del conjunto que cumplen cierta condición. Recibe una función (que dado un elemento, retorna un booleano, criterio/condición) y una lista, y retorna una lista con los elementos que cumplen con el criterio. No convierte/transforma los elementos de la lista, los filtra.

Funciones "all" y "any"



all :: (a -> Bool) -> [a] -> Bool → Cuantificador Universal

any :: (a -> Bool) -> [a] -> Bool →Cuantificador Existencial

A ambas funciones se les pasa como parámetros una función (condición) y una lista, y retornan un Booleano. All nos dice si TODOS los elementos de la lista cumplen la condición que le pasé, nos devuelve true si esto es cierto. Any, nos dice si ALGUNO de los elementos de la lista cumple con la condición que le pasé. También podría usar filter y ver si el tamaño de la lista final coincide con el de la lista inicial, o si el tamaño de la lista final es igual a 0. Pero no tiene sentido.

Siempre que tenga un problema busco la herramienta que esté más cerca de ese problema para resolverlo. No hay que distraerse buscando algoritmos si tengo herramientas que me lo resuelven.

Loop Infinito

Una función que no podría terminar nunca, ya que no hay ningún punto en el que se corte la recursividad, puede ser usada en un contexto que acote la ejecución gracias a la evaluación diferida

Listas por Compresión

Son un azúcar sintáctico que nos permite armar listas a partir de los elementos de otra luego de aplicar filtros y transformaciones.

Listas por Rangos: [1 ... 100]

Listas Infinitas: [1..]

Ctrl + c → Pare de ejecutar la consola

Definiciones Locales: Where

Definir funciones que sólo se encuentran definidas en el contexto de una función. Ejemplo:

```
imc peso altura
  | peso / altura ^ 2 <= 18.5 = "Bajo Peso"
  | peso / altura ^ 2 <= 25.0 = "Peso Esperado"
  | otherwise                 = "Alto Peso"
```

```
imc peso altura
  | indiceGordura <= 18.5 = "Bajo Peso"
  | indiceGordura <= 25.0 = "Peso Esperado"
  | otherwise             = "Peso Esperado"
    where indiceGordura = peso / altura ^ 2
```

Nota: Al igual que con guardas, para que funcione hay que dejar al menos 1 espacio de indentación

FOLD / Reducción

Indentación = Sangría

Listas en No-Tan-Alto Nivel

Las usamos cuando tenemos funciones que producen resultados a partir de todo el conjunto. Mira todo el conjunto de la lista y saca conclusiones del todo.

Fold

A partir de un conjunto, me interesa llegar a un resultado (Perspectiva de Currificación)

fold :: (a -> b -> b) -> b -> [a] -> b → fold :: (a -> b -> b) -> b -> ([a] -> b)

Le pasamos como parámetros, una función reductora, una semilla, y un conjunto (lista); para que nos retorne algo. La semilla es un resultado para el caso de que b esté vacío. La función reductora agarra un resultado parcial y un elemento y calcular el resultado (parcial) con ese elemento incluido. Cada elemento me genera un nuevo resultado parcial. Cada resultado parcial es la semilla para aplicar de nuevo la función con el siguiente elemento. Cuando "se me terminan" los elementos, obtengo el resultado final. Ejemplo: fold (+) 0 [1, 2, 3] = 6

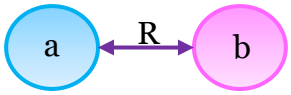
Tipos de Fold

foldr :: (a -> b -> b) -> b -> [a] -> b → foldr1 :: (a -> a -> a) -> [a] -> a : La semilla es la head

foldl :: (b -> a -> b) -> b -> [a] -> b → foldr1 :: (a -> a -> a) -> [a] -> a : La semilla es la head

La diferencia entre r y l es la dirección en que recorre la lista el fold.

Unidad n° 2: Paradigma Lógico



Se basa en los conceptos de lógica matemática. Predicados que caracterizan o relacionan a los individuos involucrados y la deducción de las posibles respuestas a una determinada consulta. **Es un paradigma declarativo** (solo contamos cuándo pasan las cosas). No hay asignaciones destructivas, se trabaja con el concepto de unificación (cuando una variable toma un valor, tiene el mismo valor en todas sus instancias). No hay tipado, ni efecto. Las relaciones son bidireccionales. Puede haber valores que no cumplan con "Existencia" o "Unicidad".

Cada programa de Prolog es una **base de conocimiento**: En ella se escribe todo lo que se sabe (SIEMPRE es VERDADERO) en forma de predicados. Se compone de cláusulas que definen predicados partiendo de los individuos de los que queremos hablar.

Principio de Universo Cerrado

El motor asume como falso todo lo que no pueda probar como verdadero (todo lo que está fuera de la base de conocimiento), no podemos demostrar su veracidad. En Funcional necesitamos poner una variable al final del Pattern Matching para que no se quede "colgado". Acá no hace falta, porque se considera como falso si no se encuentra alguna sentencia (predicado) verdadera.

Parámetros

La forma de decir que algo es falso es no decir nada.

Individuos

Son aquellas cosas sobre las que versa el conocimiento que queremos expresar. Es cualquier entidad acerca de la cual nos interese estudiar sus características o relaciones con otros individuos.

Individuos Simples

Átomos: Los átomos son valores que representan a una entidad indivisible. Son las entidades que comienzan con minúscula. No son Strings. Por ejemplo: yamila, analia, axel, leandro.
Números: Pueden usarse para lo mismo que en cualquier otro paradigma. Los podemos comparar, saber si uno es mayor que otro y usarlos para resolver operaciones aritméticas.

Individuos Compuestos

Listas: Estructuras como las que venimos viendo. se encierra entre corchetes [ ] donde cada elemento del conjunto se separa por coma
Functores: Son individuos que nos permiten agrupar otros individuos para formar una abstracción más compleja. Tienen un nombre y una aridad determinada.

Variables y Variables Incógnitas

Toman cualquier valor que se les indique, también se les llama "incógnitas". Los términos en minúscula se refieren a cosas particulares y las palabras en mayúscula son incógnitas.

Predicados

Son las cosas que queremos decir (o predicar) acerca de los individuos. Comienzan con minúscula. Entre los nombres y los paréntesis, no debe haber espacio.

Cláusulas

Sentencias/unidades de información de una base de conocimiento. Deben terminar con un punto (.) (comienza con el nombre). Un predicado puede tener o no, más de una Cláusula. Cada cláusula puede ser un hecho o una regla. La principal diferencia entre el hecho y la regla es que la regla tiene un antecedente; y el hecho no, el hecho es verdadero siempre.

Hecho

Hace una afirmación incondicional (no depende de ninguna condición para ser cierta), generalmente sobre un individuo particular. (No contiene el símbolo ":-")😊
Siempre son verdaderos. Los hechos me permiten definir **por extensión** el conjunto de individuos que tienen una característica. nombrePredicado(Parámetros).

Regla

Define una implicación, es decir que define que si se cumplen ciertas condiciones, entonces un predicado se verifica para ciertos individuos. Su valor de verdad depende de otros predicados.
Me permiten definir el conjunto de individuos que tienen una característica, **por comprensión**.

Una regla se compone de una cabeza y un cuerpo, unidos por el símbolo "**:-**" que denominamos cuello. Si vemos una regla como una implicación con antecedente y consecuente, está invertida respecto a lo que se vio al estudiar lógica: la cabeza es el consecuente, el cuerpo es el antecedente.

```
nombrePredicado(Parámetros) :-
        condicion1(ParámetrosC1), ...
        condicionN(ParámetrosCn).
```

Ejemplo:

```
humano (yamila).
humano (axel).
mortal (Alguien) :- humano (Alguien).
```

Aridad de los Predicados

Cantidad de Parámetros que tiene un predicado: nombrePredicado/ n, n>=1
nombrePredicado(P1, P2, ..., Pn)

Podrían haber predicados con el mismo nombre pero distinta aridad y serían totalmente independientes entre ellos. A partir de su aridad podemos separar los predicados en:
Propiedades: Son los predicados de aridad 1, que expresan características de individuos.
Relaciones: Son los predicados de aridad mayor a 1, que expresan relaciones entre individuos.

Consultas

Forma de usar un programa en lógico, se hace una consulta, y se obtiene una o varias respuestas.
Individuales: Se hacen sobre individuos específicos. Preguntamos si una combinación de individuos en particular hacen verdadero un predicado. Devuelven True o False.
Existenciales: Se verifican si EXISTE ALGÚN INDIVIDUO que haga verdadero al predicado. La/s variable/s por la cual/es pregunto, van con un guion bajo. Devuelve True/False
Variables: Es un tipo de preguntas existenciales. En este tipo de consultas aparecen variables o incógnitas, es un caso particular de las existenciales, donde no paso una variable anónima (una con mayúscula). Nos devuelven todos los valores que hacen verdadero al predicado.

Conjunción y Disyunción

La conjunción lógica ("y", ^), se logra con la coma (,). Mientras que la disyunción lógica ("O", v), la conseguimos mediante la definición de varias cláusulas para el mismo predicado.

Unificación vs Asignación

La asignación sucede cuando es posible reemplazar el valor de una variable por otro (No hay en lógico). Las variables en el paradigma lógico se asemejan a la idea de variable matemática, y el mecanismo por el cual se le dan valores a las variables se llama **unificación**. Cuando una variable que no tiene ningún valor pasa a tenerlo vamos a decir que dicha variable ha sido **ligada**, en caso contrario la variable se encuentra **sin ligar o no ligada.** Una variable siempre que tome un valor, va a tener siempre el mismo valor dentro del predicado/ la consulta, ya que se unifica. Pero dos variables diferentes también pueden unificarse con un mismo valor, por lo que es necesario aclarar si queremos que dos variables sean diferentes dentro de nuestro predicado. Básicamente lo que hace Prolog es buscar un consecuente (revisar al principio de todo qué era eso de consecuente) dentro de todas las cláusulas de nuestra base de conocimiento que unifique con la consulta.
La diferencia entre decir Pattern matching y unificación es bastante gris (algunos lo consideran sinónimos). Es común decir "unifica" o "matchea" indistintamente. Vamos a hablar de unificación de variables en relación al valor que las mismas toman en base a una consulta y de pattern matching cuando en el encabezado del predicado se determina la forma que debe tener.

Predicados simétricos

Un predicado es simétrico, si la relación que modela lo es: aRb => bRa

```
odia(platón, diógenes).
odia(diógenes, _).

?- odia(diógenes, X).
   true. ☹
```

Inversibilidad

Es la capacidad de lugar una variable de un predicado. → → →
Si puedo hacer una consulta existencial en una de las variables del predicado, entonces **esa variable** es inversible. Si todas las variables son inversibles, el predicado es totalmente inversible. Debo asumir que el predicado es completamente inversible, a menos que haya algo que pueda comprometer su Inversibilidad. Estos casos tienen que ver con las submetas de un predicado que requieren variables ligadas, hay que fijarse para saber qué hace falta generar. Que un predicado sea inversible significa que los argumentos de este pueden usarse tanto de entrada como de salida: no es necesario pasar el parámetro resuelto (un individuo), sino que se puede pasar una incógnita.

Inversible quiere decir “puedo hacer consultas con cualquier combinación de valores e incógnitas”, no que “si invierto los argumentos da las mismas respuestas”. Esto es simetría.

Generación

Acotar el conjunto de valores posibles para una variable, o ligar la variable con un valor, ya que eso cambia la forma en la que se evalúan las siguientes consultas. Esto último tiene que ver con los problemas de Inversibilidad, y los casos de uso se dividen en dos grandes grupos:

a) Cuando la generación posibilita la siguiente consulta.
b) Cuando la semántica de la consulta es distinta a la que quiero.

Casos Posibles de No-Inversibilidad

Hechos con Variables

Cuando un hecho tiene declarado al menos una variable/variable anónima, toma como true cualquier valor en uno (o más) parámetros que lo componen. Si hago una pregunta variable para ese parámetro, no me puede retornar los casos posibles, solo me responde “true”. Ejemplo: leGusta (pepe, _). → A pepe le gusta cualquier cosa, sin acotar qué podría ser aquello que le gusta. Para hacer el predicado inversible, deberíamos acotar qué puede ser aquello que le gusta convirtiéndolo en una regla. Ejemplo: leGusta(pepe, Comida) :- comida(Comida).

Not

Las variables que aparecen en la parte de la cláusula deben llegar ligadas, a menos que sean variables afectadas por un “para ningún”, en este caso deben llegar libres. Ejemplo: esPlantaComestible(Planta) :- not(esVenenosa(Planta)).→ No va a ser inversible, porque la variable “Planta” puede asumir cualquier valor y dar true al no ser venenosa.
En este caso, lo que hacemos es acotar los posibles valores que puede asumir “Planta”: esPlantaComestible(Planta) :- esPlanta(Planta), not(esVenenosa(Planta)).

Aritmética

Aritmética: Todas las variables a la derecha del is deben llegar ligadas al is.
siguiente (Numero, Siguiente) :- Siguiente is Numero + 1.
No es inversible para Numero, al llegar a la cuenta hay una variable no ligada a la derecha del is. En este caso no hay forma trivial de arreglarlo para que el predicado sea totalmente inversible.
siguiente (Numero, Siguiente) :- esNumero (Numero), Siguiente is Numero + 1. → Ahora es totalmente inversible, ya que reduce los casos posibles. Ejemplo: ?- siguiente(Numero, 43). → 42.
Para llegar a ese resultado, Prolog prueba uno a uno los casos posibles hasta que haya un caso posible. Eso el algo que llamamos ***backtraking***: Proceso para encontrar múltiples valores por fuerza bruta. Va a buscar todas las combinaciones posibles de elementos que satisfagan la respuesta, va a deshacer y volver para atrás.

Comparación

Lo que comparemos debe estar ligado: plantaHeavy(Planta):-Nivel > 5, nivelVeneno(Planta,Nivel).
Es incorrecta, porque Nivel no está ligado al momento de comparar. Esta forma sí es correcta: plantaHeavy(Planta) :- nivelVeneno(Planta, Nivel), Nivel > 5.

Recursividad

Si el caso base es inversible, entonces también lo serán los casos recursivos.

ForAll

Al usar este predicado, la/las variables que no son compartidas entre Universo y condición deben estar ligadas. Ejemplo: friolento(Animal):- forall(habitat(Animal, Bioma), esTemplado(Bioma)).
Si yo no ligo la variable “Animal” a “esAnimal”, no tengo forma de decir que, por ejemplo, un unicornio no es friolento; porque al no ser un animal, no habita ningún bioma que no sea templado. Entonces necesito si o si, ligar la variable “Animal” a el universo de los animales: friolento(Animal):- esAnimal(Animal), forall(habitat(Animal, Bioma), esTemplado(Bioma)).

Functores y Polimorfismo

Cuando tenemos “Accessors” a los Functores, los parámetros del predicado no son inversibles. Para lograr que lo sean, se asocian a otro predicado. Ejemplo:
autor(libro(_, Autor, _, _), Autor) :- vende(libro(_, Autor, _, _), _). →Hacemos que “Autor”
autor(cd(_, Autor, _, _, _), Autor) :- vende(cd(_, Autor, _, _, _), _). sea Inversible.

Findall

Los primeros dos parámetros no son inversibles, ya que son las condiciones a cumplir.

¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡¡SIEMPRE GENERAR CON FINDALL Y FORALL!!!!!!!!!!!!!!!!!!!!!!!!!!!!
Es necesario que se generen los parámetros aunque parezcan ligados dentro de los find/forall.

Orden Superior

Un predicado es de orden superior, si recibe una consulta (otro predicado) por parámetro. Por ejemplo, "not": esTerrestre(UnAnimal) : - hábitat(UnAnimal, _), not(esAcuatico(UnAnimal)).
Es posible construir nuestros predicados de orden superior, pero no lo vemos en la materia.
Los predicados de orden superior son: not, forall, Findall, distinct.

Cuantificador Universal

Es el "Para todo": ∀x∈X p(x) → Para todo x perteneciente a un conjunto X, se cumple p(x).
Donde tenemos el Universo y la condición.
En Prolog, utilizamos forall/2 : forall(Universo, Condición). Es un predicado de orden superior, de aridad 2, que recibe dos consultas: El Universo (Consulta existencial, que me genere valores, un universo sobre el cuál trabajar), y la condición (Consulta que me genera valores de verdad, para saber si se cumple o no esa condición para cierto átomo/valor). La forma de conectar esas consultas, es que ambas reciban una variable.

Not vs Forall

∀x∈X p(x)↔-∃ x∈X -p(x)

Ejemplo: forall(habitat(A, B), esTemplado(B)). ↔ not((habitat(A, B), not(esTemplado(B)))).
"Si todos los animales viven en hábitats templados, no hay animales que no vivan en hábitats templados".
Vamos a buscar la manera más fácil de resolver las cosas, quizá usar dos negaciones sea más complejo, pero ambos predicados son válidos. Hay resultados que son naturalmente negados, por lo que en ese caso es mejor el uso del not, pero también sería válido usar forall. Necesito un paréntesis más para usar el not principal, ya que es de aridad 1.

Consecuencias: -∀x∈X p(x) ↔ ∃ x∈X -p(x) →"Si no todos los animales viven en hábitats templados, existen animales que no viven en hábitats templados".

∀x∈X -p(x) ↔ -∃ x∈X p(x) →"Si ningún animal vive en hábitats templados, no existe animal que viva en hábitats templados.

Functores

Son estructuras compuestas que tienen nombre. Se usan para representar datos que tienen similitudes pero no comparten todos los parámetros. Por ejemplo:

vende(NombreLibro, Autor, Genero, Editorial, Precio).
vende(NombrePelicula, Director, Genero, Año, Precio).

Necesito los Functores para hacer que el predicado "vende" sea polimórfico

vende(libro(Nombre, Autor, Genero, Editorial), Precio).
vende(película(Nombre, Director, Genero, Año), Precio).

vende(Artículo, Precio).

Los ponemos en el predicado para que representen la información que queremos. No hace falta declararlos, ni generar nada especial para que existan, al igual que los predicados.
La sintaxis es igual a la de los predicados, de hecho, cuando usamos un predicado de orden superior, recibe un functor como parámetro, al cual decide tratarlo como una consulta.
La diferencia nace de que al functor siempre van a estar como parámetros de los predicados, nunca puedo definir o usar un functor a nivel raíz. **No** son predicados, no tienen un valor de verdad.

SOLO PUEDO HACER CONSULTAS SOBRE PREDICADOS → No voy a poder consultar algo de un functor nada más, sino del predicado que lo contiene.

Para usar un parámetro específico del artículo, que aparece en todos los Functores, puedo abstraerme: Uso un predicado auxiliar que me relacione a la estructura con ese parámetro.
Ejemplo: autor(libro(_, Autor, _, _), Autor). → Me devuelve el autor de lo que se vende,
autor(cd(_ Autor, _, _, _), Autor). aunque sea de un libro o un cd.

Polimorfismo en Lógico

Puedo construir predicados que no necesitan detenerse a pensar cómo están compuestos. Ejemplo: Como el predicado "autor" (Predicado polimórfico) sabe relacionar el autor con un libro o un cd, podemos decir que trata a los libros y cd de manera polimórfica y podemos usarlo en cualquiera de esos contextos aunque no sepamos cual es. Hay muchos problemas de Inversibilidad al usar estos "Accesors", podemos volverlos inversibles para el parámetro que usemos:

```
autor(libro(_, Autor, _, _), Autor) :- vende(libro( _, Autor, _, _), _).
autor(cd(_, Autor, _, _, _), Autor) :- vende(cd(_, Autor, _, _, _), _).
```

Los predicados pueden recibir cualquier tipo de argumento, pero el uso que se hace de ellos en el interior de las cláusulas que lo definen, delimita un rango de tipos que tiene sentido recibir. No sucede un error si se recibe un dato de tipo diferente a lo esperado, sino que la consulta falla por no poder unificar el valor recibido con lo esperado, descartándolo como posible respuesta.

Listas

Conjunto de elementos que poseen un orden y que se asocian a una implementación inmutable.

Tiene dos constructores: $[a] = \begin{cases} [\,] --Nil :: [a], & es\ la\ lista\ vacía\ de\ tipo\ a. Caso\ semilla \\ (|) --Cons :: a \to [a] \to [\,], & recibe\ un\ elemento\ y\ una\ lista \\ & cabeza \to cola \end{cases}$

Son recursivas (compuestas por una cabeza y una cola que a su vez es una lista). La lista es una estructura vacía o una que tiene una cabeza y una cola (Usamos un pipe para separarlas en lógico, no dos puntos). Si la lista no está vacía, podemos dividirla en cabeza y cola tantas veces como sea necesario, hasta tener una lista vacía. Como acá no existen los tipos, puedo poner cualquier combinación de átomos en ellas. Me sirve para cuando necesito agrupar átomos.

Predicados de Listas (Alto Nivel)

member(Elemento, Lista).	length(Lista, Tamaño).	sumlist(Lista, Total).

Estos predicados no son del todo inversibles para el parámetro "Lista", ni tampoco son del todo inversibles. Prolog nos trata de devolver una lista que tome una forma parecida a la que le pido. Me devuelve listas con "PUNTEROS". Es decir, me devuelve listas que son válidas al reemplazar los punteros por valores, o todos los casos posibles. **Por lo que son más o menos inversibles... va a depender del contexto y de la información que dé en la consulta.** No puedo decir antes de saber qué consulta haré si es inversible o no. Podemos decir que los predicados no van a ser inversibles para la lista a menos que pueda calcular los miembros exactos de la lista en algún lugar.

```
?- length(L, T).
L = [],
T = 0 ;
L = [_5774],
T = 1 ;
L = [_5774, _6806],
T = 2 .

?- member(E, L).
L = [E|_8616] ;
L = [_8614, E|_9346] .

?- length(L, 2), member(a, L), member(b, L).
L = [a, b] ;
L = [b, a] ;
false.

?- sumlist(L, T).
L = [],
T = 0 ;
ERROR: Arguments are not sufficiently instantiated
```

FindAll

Findall(Selector, Consulta, Lista).

```
listaDeRecetas(Recetas):-
    findall(Nombre, receta(Nombre, _), Recetas).
```

Selector: Nos permite "elegir" qué porción de la respuesta voy a querer generar. Es una consulta.
Consulta: Es de tipo existencial. Es el que genera el universo para la lista.
Lo usamos para agrupar todas las respuestas de una consulta, para cuando necesitemos usar todas al mismo tiempo, como para contar, sumar, etc. Cumple el rol de map y filter si lo necesitamos.
No es inversible para sus primeros dos parámetros.

Is

Se usa para resolver cuentas en Prolog. Solo podemos poner la cuenta a la derecha del "is", y además, es inversible sólo para el primer parámetro.
Errores al usar Is: Usar el = en vez de el Is. Tratar de acumular variables (ejemplo: E is E+1). Usar is para verificar igualdad de valores, ya que es sólo para operaciones aritméticas: Si bien funciona, es un error conceptual, porque si dos individuos deben ser el mismo, entonces alcanza con escribirlos con la misma variable. Usar is para resolver ecuaciones (ejemplo: 3 is X+1).

Unidad n° 3: Paradigma Orientado a Objetos – Wollok

Tenemos efecto y herramientas para controlar el flujo de datos. Tenemos que diseñar teniendo en cuenta si quiero destruir o consultar. Es menos declarativo, cuidado.

En otros Lenguajes, tendemos a trabajar por un lado con Datos (una pieza de programación), y por otro con Lógica (Operaciones, trabajar/manipular datos). Un programa basado en el Paradigma Orientado a Objetos es un conjunto de objetos (Unidad de Lógica autocontenida) que interactúan entre sí en un ambiente mandándose mensajes para lograr un objetivo determinado.

Objeto

Ente computacional. No nos van a interesar tanto los objetos por sí mismos, sino que estos pueden resolver problemas, de hacer cosas: tienen **comportamiento, exponen una Interfaz.** Aunque dos objetos se vean muy parecidos, siguen siendo dos objetos diferentes: Los objetos se pueden diferenciar, porque tienen **identidad** propia. Los objetos pueden tener un **estado,** y el mismo puede cambiar. Pueden ser, o no, tangibles. Por lo tanto, podemos decir que un objeto es algo que tiene identidad, puede tener un estado interno y exponen una Interfaz (comportamiento).

Interfaz

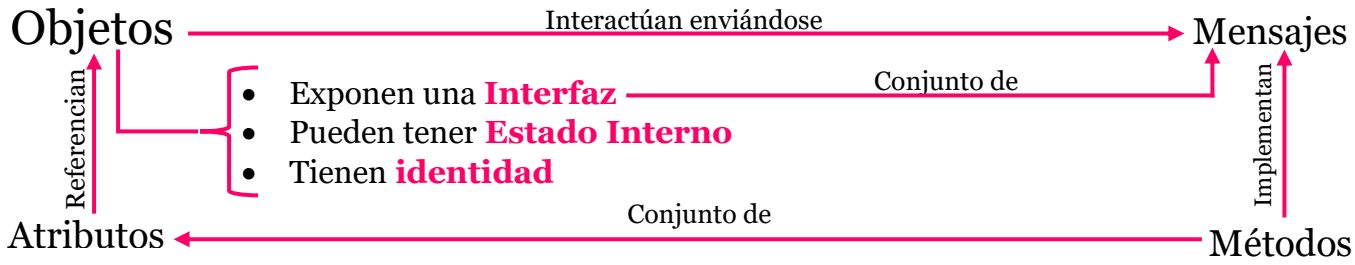
Lo que nos va a interesar es ver a un objeto desde un punto de vista externo (cuando mandamos mensajes) y desde un punto de vista interno (cuando implementamos el comportamiento que queremos). Al punto de vista externo, que sólo incluye el comportamiento que exhibe (los mensajes que entiende) le vamos a decir interfaz: Conjunto de operaciones o cosas que puedo hacer con un Objeto, a estos vamos a llamarles mensajes; es entonces un **conjunto de mensajes**, que son la única manera que tengo para interactuar con un objeto. Yo no puedo ver los datos que tiene un objeto, sino interactuar con él mediante mensajes. Definir las interfaces de un objeto, va a implicar definir qué cosas se podrán hacer en nuestro programa y qué cosas no.
Los mensajes que un objeto entiende son resultado de poseer métodos (Operaciones que saben resolver los objetos cuando les llega un mensaje), los métodos son la implementación de mensajes.

Estado Interno

Todo objeto debe exponer una interfaz, pero pueden o no tener atributos, que son referencias a otros objetos ("Puntero de Alto Nivel"), es un nombre por el cual un objeto conoce a otro objeto. Los atributos son (o pueden ser) mutables. Tenemos la capacidad de asignar destructivamente.

Identidad

Un objeto es completamente diferente a otro, a pesar de su interfaz o estado interno.



Referencias

En objetos una variable es una referencia a un objeto. La mayoría de las referencias pueden ser reapuntadas a otros objetos mediante la operación de asignación.

- Atributos: Se usan para que el objeto mantenga un estado propio.
- Locales: Se declaran dentro de un método, sólo son visibles desde él, no sobreviven su ejecución
- Parámetros: Parte de la firma del método, no pueden apuntar a otros objetos dentro del método.

Asignación de Variables

Se logra así: variable = expresionQueDevuelveUnObjeto, y debe interpretarse que cuando se evalúa esta línea, la variable referencia al objeto resultado de la expresión de la derecha: Al asignar una variable se cambia el objeto al que está apuntando esa referencia.

Var y const

En Wollok, las referencias pueden declararse como variables (con la palabra reservada var) o como

constantes (con la palabra reservada const). Las constantes están pensadas para ser usadas cuando que no se espere que la referencia pueda cambiar de valor. Es importante entender que si tengo un objeto que es una constante, puedo cambiar **su estado interno,** pero no al objeto que referencia.

Errores Comunes al Trabajar con Atributos

- Atributos redundantes: Si cierta pieza de información puede ser calculada a partir de otra, no se usa un atributo que tenga que ser mantenido consistente, sino un método que calcule lo que necesario a partir de la otra información disponible.
- Atributos innecesarios: No hay que perder de vista que nuestro programa es un modelo, y por ende no hace falta que el objeto recuerde información que luego no va a ser usada.
- Uso de atributos en vez de locales: Si necesitamos recordar cierta información sólo dentro de la ejecución de un método, lo correcto es usar una local, no un atributo.

Como regla general para evitar estos problemas: ¿Hay alguna forma de no usar un atributo para lo que estás haciendo? Si la hay, no se usa un atributo.

Pseudo Variables

Una pseudo variable es un variable manejada por el ambiente, dicha variable no puede ser asignada a otro valor. En Wollok tenemos a self (También a super, pero no es una pseudo variable)

Inicialización

Un problema común que suele darse con atributos de los objetos es olvidarse de inicializarlos: para poder usar una variable de forma razonable, la misma debería referenciar a un objeto que entienda los mensajes que esperamos mandarle, de lo contrario se generará un error.

Mensajes y Métodos

Una entidad de mi programa es capaz de mandar mensajes a un objeto, que reacciona haciendo lo que sabe hacer cuando le mandan el mensaje, si lo conoce. Si no, no va a saber qué hacer y fallará. Cuando se le envía un mensaje a un objeto, se activa un método cuyo nombre coincide con el del mensaje. Si entendemos cómo interactuar con un objeto, sabemos cómo funciona el programa.

- Un mensaje es algo que yo le puedo decir a un objeto.
- Un método es una secuencia de líneas de código que tiene un nombre.

- Los mensajes los entienden los objetos
- Si a un objeto que entiende el mensaje **m** le envío el mensaje **m** entonces se va a activar el método **m** para ese objeto
- Cuando en un método dice self, se referencia al objeto que recibió el mensaje.

A quiénes se le pueden mandar mensajes: A los objetos conocidos mediante atributos, a los que me pasan por parámetro, a los bien conocidos (incluyendo a los literales), y a mí mismo (self).

Tipos de métodos y mensajes

```
object pepita {
  var energía = 100
  method volar(kms){
    energía = energía - kms *2 }
  method comer(grs){
    energía = energía + grs * 9 }
  method energía() = energía
}
```

pepita es una definición declarativa, no una secuencia que se ejecuta. Contamos las características del objeto en términos de sus atributos y métodos. La variable energía está inicializada en 100. Los métodos son de dos tipos: Con cuerpo (entre corchetes) y sin cuerpo. Esto nos permite saber cuándo los usamos con efecto y cuando de forma declarativa.

Buscamos separar las operaciones que tienen el fin de causar un efecto (ordenes, instrucciones) y las consultas (retornos).

Los métodos que están definidos por un igual siempre retornan un atributo, los otros, causan efecto. Estas son las dos posibilidades que tenemos de hacer mensajes. En las operaciones que causan efecto, causamos una asignación destructiva. Debemos usarla de manera controlada.

Pilares de la Programación Orientada a Objetos

Hay ciertas cuestiones básicas que debemos tener en cuenta. Vamos a nombrar a las que se destacan en este paradigma en contraposición de los que ya conocemos.

Encapsulamiento

Un objeto está encapsulado, porque nos oculta y protege detalles de su programación; porque la capa que yo veo del objeto es su interfaz, la necesidad de conocer del objeto sus detalles de implementación es mínima: Me interesa qué puede hacer, resuelve las cosas como una caja negra.

Yo no puedo hacer cosas que el objeto no me permite (no manipulo cosas erróneas).
Quien usa un objeto sólo ve lo que necesita para interactuar con él: el comportamiento que exhibe. Los detalles internos quedan encapsulados en el objeto. Así quedan acotadas y explicitadas las formas posibles de interactuar con él. Permite separar mejor las responsabilidades y evitar efectos inesperados como resultado de la modificación del valor de las variables por entidades externas.

Delegación y Responsabilidades

Responsabilidad

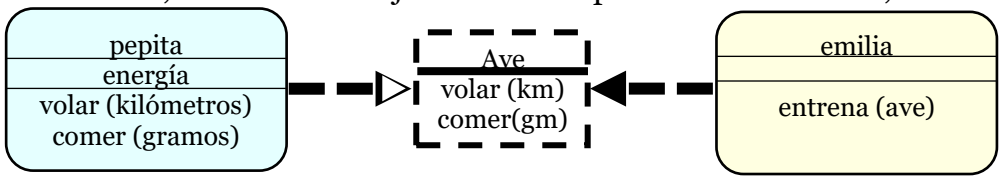
Aquello que hace un objeto, que nosotros queremos que haga. Si tenemos un solo objeto que hace demasiadas o todas las tareas de nuestro programa, probablemente sea complicado que nuestro programa cambie y resuelva nuevos problemas, saber si funciona correctamente (testearlo), y arreglar los errores que tenga (si los tiene). Si tenemos varios objetos con menos responsabilidades cada uno, va a ser más sencillo intercambiarlos usando polimorfismo, reutilizar partes de código en otras partes de nuestro programa, evitar repeticiones de lógica...

Delegación y Colaboración

Es la forma de repartir las responsabilidades → delego responsabilidades a otros objetos.
Los objetos se conocen a través de referencias, y así pueden mandarse mensajes. Cuando un objeto resuelve parte de un problema y le pasa otra parte del problema a algún objeto que conozca, hablamos que ambos objetos están colaborando. Cuando un objeto le encarga toda la tarea a resolver a otro objeto, decimos que este delega la responsabilidad.

Polimorfismo

El polimorfismo en este paradigma se define como la capacidad que tiene un objeto de poder tratar indistintamente a otros que sean potencialmente distintos. Cuando hablamos de que un objeto "trate" a otros, estamos hablando de que interactúen a través de mensajes. Van a ser necesarios como mínimo 3 objetos para que ocurra el polimorfismo, 1 que usa, y dos que son usados. Y es necesario que un tercero quiera usar a otros dos objetos de manera indistinta, y que estos entiendan los mensajes que ese tercero quiere enviarles. ¡Es la herramienta más importante!
Entonces, podemos decir que un objeto trata polimórficamente a otros cuando les envía a ambos exactamente los mismos mensajes, sin importarle cual es cual y ellos pueden responder ya que tienen una **interfaz común**. Para que dos objetos sean polimórficos en un contexto determinado (es decir, ante el tercero, en un momento particular), es condición necesaria que entiendan los mismos mensajes. Sin embargo, para que puedan ser usados indistintamente, además de ello, deben comportarse ante tal envío de mensaje, de una forma similar desde el punto de vista del dominio, produciendo efectos similares, devolviendo objetos también polimórficos entre sí, etc.
Ejemplo: Emilia no necesita que pepita sea un ave, sino que sepa volar y comer. Al ser "Ave" la interfaz de la que deriva pepita y a la que se relaciona emilia, podríamos tener otras aves (o un perro).



Esto quiere decir, que si aparecen "nuevas aves" o "nuevos entrenadores", no necesito parchar todo el código y los objetos, solo necesitan tener las interfaces (seguir las reglas) para usarse.

Getters y Setters

El uso de setters y getters (mensajes para modificar y conocer el valor de un atributo respectivamente), también conocidos como accessors, es importante para que el objeto que tiene esos atributos pueda controlar el uso de estos y para que los que usan al objeto que los tiene no sufran un impacto muy grande si la implementación de este cambia.

- Getter: method nombreVariable() = nombreVariable
- Setter: method nombreVariable(nuevoValor){nombreVariable = nuevoValor}

Azúcar Sintáctico: Pueden usarse sin escribir los métodos: var property nombreVariable =...

Clases

Las clases NO SON OBJETOS en Wollok o Java, hay otros lenguajes en los que sí (Smalltalk, Ruby)
Una clase sirve de fábrica de objetos. Modela las **abstracciones** de mi dominio (los conceptos que nos interesan), permitiéndome definir el **comportamiento** y **los atributos** de las instancias.

OBJETOS ≠ CLASE → Permiten instanciar objetos, definen atributos y proveen métodos.
El nombre debe estar relacionado a lo que representa. Ejemplo: Camión (1), Camiones (conjunto)

¿Por qué necesito clases?

Porque tengo varios objetos sospechosamente parecidos. Es decir,

- El comportamiento de varios objetos es igual. No solo entienden los mismos mensajes, sino que tienen los mismos métodos (exactamente el mismo código).
- sus atributos son los mismos, pero el estado interno es diferente. Si bien tienen los mismos atributos, éstos pueden apuntar a diferentes objetos.
- su identidad es diferente. Incluso si se comportan igual y se encuentran en el mismo estado.

En conclusión, necesitamos una abstracción donde pongamos el código y los atributos en común de todos estos objetos: La clase; y cada objeto que se comporte igual va a ser una instancia de esta. Podemos pensar a las clases como "Especies" y a las instancias como "individuos" de esas especies.

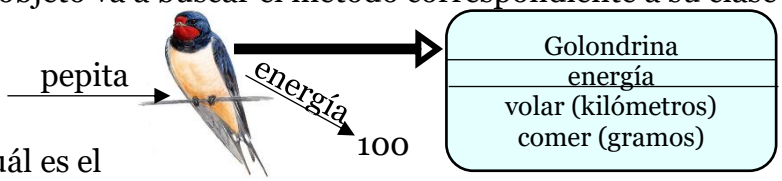
Cosas a recordar

```
class Golondrina {
  var energía =100
.....}
```

- Todo objeto es siempre instancia de una y sólo una clase.
- No se puede cambiar la clase de un objeto una vez creado.
- Los métodos y atributos declarados en una clase son para sus instancias.

Method Lookup

Si el código está en la clase, entonces, ¿Cómo responde ahora un objeto a un mensaje? Con el Method Lookup: Mecanismo por el cual un objeto va a buscar el método correspondiente a su clase (si no lo encuentra, no lo entiende).



Instanciación

Llamamos instanciación a la creación de un objeto a partir de una clase, la cual define cuál es el comportamiento y los atributos que debería tener dicha instancia. Necesitamos crear las instancias a partir de la clase (Ejemplo, creamos la instancia "Simba" a partir de la clase "León"). O sea que la clase no sirve sólo para definir el comportamiento y los atributos, sino que también sirve para crear los objetos que luego usaremos en nuestro programa. ¡No se puede cambiar de clase!
Si un objeto instanciado entiende un mensaje, siempre lo hará, ya que será consistente.

Herramientas de Instanciación

Inicialización directa

Permite crear instancias de una clase parametrizando los valores iniciales para los atributos que correspondan usando parámetros nombrados. Podemos crear una instancia de la clase Golondrina manteniendo el default para la energía de esta forma: new Golondrina(). Si nos interesara crear una golondrina con 25 de energía, la inicialización directa permite parametrizar esta información de esta forma: new Golondrina(energía = 25), y el default simplemente no se usará.
En caso de haber varios atributos, pueden inicializarse tantos como queramos, por ejemplo: new Dirección(calle = "Amenábar", numeración = 140). No importa el orden en el cual se indiquen estos parámetros, solamente debe coincidir con el nombre del atributo que se está inicializando.

Constructores

Son métodos especiales cuya finalidad es la creación de objetos de la clase indicada con la inicialización de estado interno correspondiente. Para invocarlos se usa la palabra reservada new seguido por el nombre de la clase y los parámetros según corresponda. Permite definir más de un constructor con distinta cantidad de parámetros, de modo que se permita al usuario elegir la forma más adecuada para crear las instancias. Esto permite que siempre que obtengamos un objeto, el mismo ya esté en condiciones de ser usado en vez de requerir que el usuario recuerde indicar los valores necesarios uno por uno mediante setters, teniendo mientras tanto un objeto con un estado inválido que no puede ser usado.

Garbage Collector

Si una instancia no es referenciada desde ningún lado, la misma ya no podrá ser usada, ya que nadie va a poder mandarle mensajes. Sin embargo no debemos preocuparnos por la memoria que ocupen estos objetos no olvidados, ya que ese trabajo es del Garbage Collector: Mecanismo que provee la Máquina Virtual para manejar la memoria de forma transparente para el desarrollador. Cuando un objeto ya no es referenciado por ningún otro, deja de ser útil porque nadie puede

mandarle mensajes. Si ya no es útil, está ocupando espacio innecesariamente, con lo cual el Garbage Collector se encarga de liberar ese espacio sin afectar al sistema. Podemos evitarnos los problemas asociados al manejo manual de memoria que ocurren comúnmente, y nos permite concentrarnos en los que nuestro programa debe hacer con un grado de abstracción más alto.

Si no almaceno los objetos, no tengo manera de recuperarlo → se lo come el Garbage Collector.

Igualdad, Identidad, Inmutabilidad

Hay que elegir, al diseñar nuestro programa, si con nuestras operaciones queremos o no causar efecto/si queremos o no asignar destructivamente.

Inmutabilidad: Como se dijo en "Conceptos Transversales", la inmutabilidad es una decisión de diseño en los Paradigmas con efecto colateral. Decimos que un objeto es inmutable si no puede cambiar su estado interno (su conjunto de atributos) después de su inicialización.

Es importante tener en cuenta que no alcanza en algunos casos con que el objeto no exhiba comportamiento que permita mutarlo, ya que podría tener un atributo que referencie a un objeto que sí es mutable. Luego si alguien le manda un mensaje que retorna a ese objeto mutable, y a ese objeto le manda un mensaje que lo modifica, el objeto inicial habrá sufrido cambios.

Si estamos haciendo nuestros propios objetos inmutables, hay que tener en cuenta si los objetos que conocen pueden o no cambiar su estado. En el caso de que no, listo, pero si sí pueden, hay que plantearse si dichos objetos no deberían ser también inmutables, y en el caso de no querer que así sea, retornar copias de estos cuando sea necesario.

Identidad: decimos que dos objetos son idénticos si son el mismo objeto. Dentro del ambiente podemos tener dos referencias diferentes al mismo objeto. En Wollok el operador usado para comparar dos objetos por identidad es ===. >"hola " + "mundo" === "hola mundo" => false.

Igualdad/equivalencia: por defecto 2 objetos son iguales si son idénticos. La igualdad puede ser redefinida con una implementación específica si el problema lo amerita. El operador usado para comparar dos objetos por igualdad es ==. >"hola " + "mundo" == "hola mundo" => true.

Importante: Sólo debería redefinirse la igualdad basado en valores que no vayan a cambiar.

Otra cosa sobre los objetos inmutables es que la igualdad ya no se basa en la identidad. Debería ser cierto por ejemplo que 'hola' sea igual a 'ho' concatenado con 'la', independientemente de que sean o no el mismo objeto. La igualdad termina dependiendo de los valores de sus atributos (o un subconjunto de ellos). Por eso es muy común redefinir la igualdad para los objetos inmutables.

No es lo mismo decir que un conjunto es inmutable a que su referencia es constante. En una constante, yo no puedo redirigir la referencia, pero sí puedo mutar a lo que apunta, si ese objeto tiene referencias mutables. Hay que "planear para el efecto"

Colecciones

Nos permiten agrupar objetos, para luego poder operar sobre un elemento en particular, sobre algunos elementos seleccionados mediante un filtro, o sobre una colección como conjunto. Es correcto pensar que son conjuntos de objetos, pero es más preciso pensarla como un **conjunto de referencias:** los elementos no están adentro de la colección, sino que la colección los conoce, la colección no tiene "adentro suyo" a sus elementos. Las colecciones también son objetos.

Cosas que podemos hacer con una colección: Ordenarla, Recorrerla y hacer algo con cada elemento, Agregar/Quitar elementos, Filtrar o Seleccionar según algún criterio, Preguntar si hay algún elemento que corresponde con un criterio.

Sabores de Colecciones en WOLLOK

En Wollok disponemos de dos sabores básicos de colecciones: las listas y los sets (conjunto matemático). Se diferencian principalmente por las siguientes características:

- Las listas tienen orden, los sets no, por eso sólo es posible obtener un determinado elemento en base a su posición sólo si es una lista con el mensaje get(posicionBaseCero)
- Los sets no admiten repetidos, las listas sí, por eso si se agrega un elemento repetido a un set no se agregará una nueva referencia al mismo, mientras que en las listas se incluirá una nueva referencia (en otra posición) **al mismo objeto.**

Literal de listas: [1,2,3]	Literal de sets: #{1,2,3}

Diccionarios

Conjunto de asociaciones entre claves y valores, donde tanto las claves como los valores son objetos cualesquiera.

Colecciones y referencias

Los elementos de una colección son objetos como cualesquiera otros, al agregarlos a una colección estoy agregando una referencia que parte de la colección y llega al objeto "agregado". Los objetos que elijo agregar una colección pueden estar referenciados por cualesquiera otros objetos.
La colección maneja referencias a los elementos que le voy agregando (p.ej. enviándole el mensaje add(elemento)), análogas a las referencias de las variables de instancia de otros objetos. Hay dos diferencias entre las referencias que mantiene una colección y las que mantienen nuestros objetos:

- Las referencia que usan nuestros objetos tienen un nombre, que es el nombre que luego usará para hablarle al objeto referenciado; las de la colección son anónimas.
- Nuestros objetos tienen una cantidad fija de referencias, una colección puede tener una cantidad arbitraria, que puede crecer a medida que le agrego elementos.

Los objetos que quedan referenciados por la colección pueden tener otras referencias. Un objeto no tiene nada especial por ser elemento de una colección, sólo tiene una referencia más hacia él. Un objeto no conoce de qué colecciones es elemento. La referencia a un objeto por ser elemento de una colección cuenta para que el objeto no salga del ambiente cuando pasa el Garbage Collector.

Bloques

Bloque
apply(args)

Son interfaces que consisten en el método apply(arg), que retorna "algo":

```
class Vaca{
  method estaContenta( )=...
  method litrosDeLeche( )=...
  method ordeñar(){...}
}
object criterioEstaContenta{
  method apply(vaca) = vaca.estaContenta()
}
```

```
>vacas.filter(criterioEstaContenta)
```

El filter (así como otras operaciones de listas), espera un mensaje del tipo apply. Es mala esta aplicación ya que no quiero crear objetos.
Azúcar Sintáctico: {vaca => vaca.estaContenta()}
Son una forma parecida a una función anónima, para construir objetos que entiende la interfaz completa. Es "parecido" al orden superior.
Los vamos a usar más que nada en operaciones donde necesitamos algo parecido a una función lamda, pero como solo podemos pasar objetos por parámetro, usamos Bloques
En resumen: cuando quiero hacer una operación sobre una colección que necesita enviarle mensajes a cada elemento, la operación se la pido a la colección, y le voy a enviar como parámetro un bloque que describe la interacción con cada elemento.
Un bloque (closure) es un objeto, y por lo tanto podemos hacer con él lo mismo que hacíamos con los demás objetos esto es: Enviarle mensajes, Pasarlo como parámetro de algún mensaje, Hacer que una variable lo referencie. Un objeto bloque representa un cacho de código que no se ejecutó, ese código solo se ejecutará si alguien le manda un mensaje que le indique hacerlo. Por ejemplo: Si ejecutamos las siguientes instrucciones: var x = 0, {x = x+1}. La variable "x" seguirá apuntando al objeto cero (0). Para que apunte a 1, debemos hacer que el bloque se ejecute: {x = x+1}.apply( )
apply() es un mensaje que le llega al objeto bloque. Este mensaje hace que se ejecute el código que está dentro y que se retorne el objeto devuelto por la última sentencia de este. Otro Ejemplo:

method irYVolver(metros, veces){
  veces.times
  ({n => self.volar(metros * 2) })
}
	El mensaje times(..) se encarga de mandarle el mensaje apply( ) al bloque que le pasamos. Dado que la referencia metros existía en el contexto en el cual ese bloque fue creado, es válido usarla dentro de la lógica del bloque.

También se puede ver a los bloques como objetos que representan una función sin nombre (o sea, una función anónima, como las Expresiones lambda de funcional). Ejemplo: Para representar la función g(x, y) = x^2 + 2y → var g = {x, y => x **2 + y *2}, y para usar esos bloques, por ejemplo podemos hacer: g(4, 3) → g.apply(4, 3)

Mensajes de Colecciones → WOLLOK

Con efecto

Estos mensajes alteran a la colección que recibe el mensaje. Si no quiero perder la colección original puedo o bien copiar la colección mandándole shallowCopy y luego usar estos mensajes, o replantearme si no quería usar alguno de los mensajes de colecciones sin efecto :)

colección.add(elem)	colec.addAll(otraColec)	colec.remove(elem)	colec.removeAll(otraColec)

Ordenar listas por criterio: sortBy(criterio)
Mapear con efecto/iteración: forEach(condición)

Sin Efecto

Saber cuántos tiene o si tiene un elemento: colección.size() | colección.contains(elemento)
Retornar una concatenación del sabor de la receptora: colección + otraColección
Filtrar/Seleccionar elementos: colección.filter({unElemento => unElemento.condición()})
El Filter lo que hará, será crear una lista de referencias que apunten a los mismos elementos que la originar, pero que cumplan la condición. Crea más referencias a los objetos.
Mappear: colección.map({unElemento => unElemento.condición()})
Encontrar un elemento que cumpla una condición, si no lo encuentra, explota: colección.find({...})
Ídem, pero le digo qué hacer si no encuentra uno: colec.findOrElse({elem=>elem.cond()}, {...})
All y Any: colec.all({elem=>elem.cond()})| colec.any({elem=>elem.cond()})
Foldear: colección.fold(semilla, {devolución, unElemento => unElemento.....devoluvión....})
Folders de alto nivel: colección.sum({unElemento=>unElemento.condición()})
colección.max({unElemento=>unElemento.condición()})
Obtener una Colección de otro sabor: asSet() y asList()
Ordenar listas por criterio: sortedBy(criterio)

Mensajes Para los sets

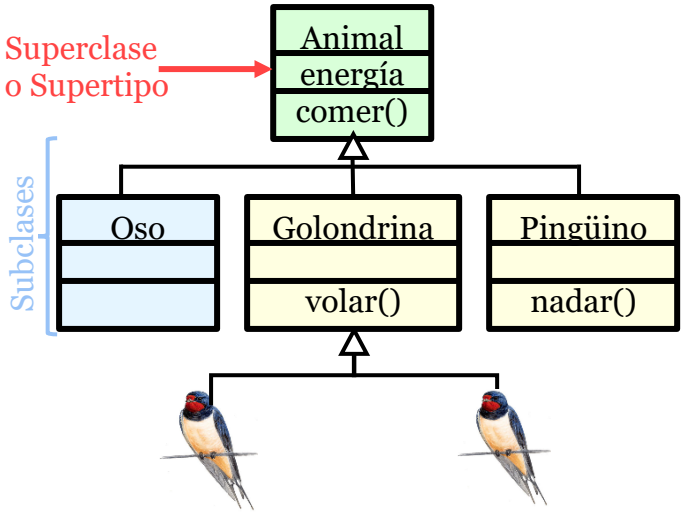
Unión: Colección con la unión de ambas: colección.union(otraColección)
Intersección: Colección con la intersección de ambas: colección. Intersection(otraColección)

Herencia

Mecanismo que permite compartir lógica/código similar. Lleva a evitar la duplicación de lógica o código. Si un objeto recibe un mensaje, mediante **Method lookup** buscará el comportamiento requerido en la clase de la cual es instancia y, si no tiene un método para él, **en sus superclases**, así hasta llegar a la raíz, y si no lo encuentra, el método no existe.

- Herencia Simple: Una clase tiene siempre una superclase pero solo una.
- Herencia Múltiple: Una clase puede tener más de una superclase. (No la usamos en Wollok)

Herencia Simple



Cuando tengo múltiples objetos con características similares, para no repetir lógica/código puedo agruparlos en clases, haciendo que los objetos le pregunten por sus métodos y atributos a su clase. Ese mismo razonamiento puedo aplicar cuando tengo múltiples clases con características similares. Puedo darles a las clases un lugar al que ir a buscar esas cosas para no repetir lógica/código, las agrupo en otras clases (super clases). Lo que hacemos entonces, es crear una relación entre clases, a la cual llamamos HERENCIA. Cuando trabajamos en un modelo de clases con herencia, conformamos un árbol, una conexión entre las distintas clases donde una va a ser la superclase y las que heredan, las subclases. Las subclases tienen una conexión muy estrecha con la superclase y van a usarla como repositorio de código, que de otra manera estaría repetido. Vamos a pensar en la superclase como una idea o una generalización que engloba algo más genérico que una clase. Lo que nos permite generalización utilizando herencia, es crear nuevas abstracciones. La premisa inicial es que siempre tengo una superclase, aunque no esté de manera explícita, la raíz de todas las clases es la clase "Object", que es la clase de la que todas las demás clases y objetos heredan comportamiento, es por lo que nuestros objetos pueden resolver lógica que entienden todos, ya que nos la estuvo proveyendo la clase Object.
Si yo creo un nuevo método en una clase, va a afectar a todas las clases y objetos que hereden de ella de manera directa o indirecta (de esa clase para abajo). Una clase es mi superclase para toda la vida, no puedo cambiarla una vez creada. En Wollok, lo ponemos directamente en la definición de las clases, usando la palabra clave inherits: class Animal {...} , class Golondrina inherits Animal{...}

Modelo una superclase porque tengo métodos/ lógica/código repetida entre varias clases.

Clase Abstracta

Es aquella que *no tiene sentido instanciar* porque es demasiado genérica y no tiene una implementación concreta para algunos mensajes que debería entender porque está definido en sus subclases. Por ejemplo, no tiene sentido instanciar la clase Object o Animal, pero sí Golondrina. Yo debo elegir qué clases debería instanciar y qué clases no, teniendo en cuenta el diseño.

No debemos asumir que todas las clases que necesitan ser polimórficas deben tener una superclase común → Solo puedo heredar de un lugar, que debe ser el más conveniente. Seguimos teniendo la idea de interfaz, y el polimorfismo se da por interfaces, no por superclaseado.

Método Abstracto

Método que necesito definir en una clase madre porque sé que usaré esa lógica, pero lo defino en las clases hijas. Esto se logra usando el método sin llaves ni igual, ejemplo: method causarEfecto(). También con "abstract". **Solo puedo tener un método abstracto en una clase abstracta.**

Redefinición

La herencia no siempre se usa para generalizar casos. A veces se usa para generar un caso particular de una clase. Si necesito que ese caso particular utilice todos los métodos de su clase madre pero con uno diferente, debo redefinirlo para que el Method Lookup lo intercepte en esa clase y no en la madre. Esto se logra con la palabra "override". Pero muchas veces no tengo cambios totales, no quiero solamente "pisar" un método, sino que quiero agregarle cosas y luego usar la lógica que hubiera heredado, puedo solucionar esto con la palabra "super".

```
class Misil{
   const potencia
   var property agotada = false
   method dispararA(objetivo){
       agotada = true
       objetivo.sufrirDanio(potencia)
}}
```

```
class MisilTermico inherits Misil{
      override method dispararA(objetivo){
            if(objetivo.emiteCalor()){
                  agotada = true
                  objetivo.sufrirDanio(potencia)
                  super(objetivo)
}}}
```

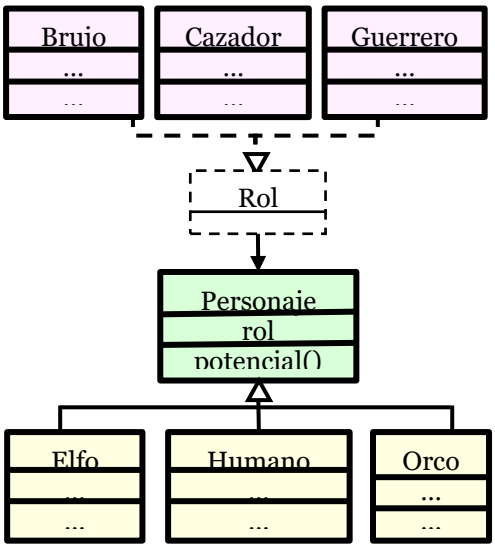
La palabra super, parece un envío de mensaje pero en realidad no lo es, no hay receptor. Es en realidad una indicación al ejecutor del código para que continúe el Method Lookup y encuentre el método que se hubiera ejecutado. Super sólo me va a servir para métodos con override. Lo que me permite esto, es evitar la repetición de lógica y código en ambos métodos.

Nota 1: Un método que sólo llama a super, NO SIRVE, solo "jode" la herencia, no tiene sentido.

Nota 2: C hereda de B y B de A. Quiero usar un método de C que usa un self y apunta a un método que se describe por primera vez en B, entonces uso ese método. Si ahora ese método usa self y usa un método que primero se describe en C, entonces uso ese método. Siempre que quiero usar un método que contiene un self, debo volver a chequear desde la clase de partida si existe ese método.

Nota 3: Para decidir si debiera o no hacer una herencia, puedo preguntarme:
1)¿Tiene sentido que una clase sea una herencia de otra clase?
2)Si el día de mañana la clase madre cambia su comportamiento, ¿quiero que la hija lo haga?

Herencia vs Composición



La composición en objetos es una relación de conocimiento entre dos objetos donde el objeto conocido puede cambiarse por otro que sea polimórfico para el que los conoce. El uso de composición en es una solución muy elegante para problemas aparejados por la Herencia:

- **Repetición de código:** La forma de subclasificar a una clase por dos categorías diferentes hace que tengamos código repetido entre las hojas del árbol de herencia. Trae problemas, sobre todo si el sistema sigue creciendo de esta forma, la repetición de código es exponencial y realizar un cambio en la lógica es difícil (problemas de extensibilidad).
- **Problemas con la identidad de los objetos:** Un objeto de una subclase no puede cambiar a otra, su clase es algo que se mantiene durante toda la vida del objeto. Si tratamos de emular el cambio de clase creando un nuevo objeto y copiando los valores de sus

atributos según corresponda, ya no será el mismo objeto para el sistema. En el Paradigma Orientado a Objetos una de las características de un objeto es su identidad, la cual estaríamos perdiendo si tomamos esa decisión y el problema asociado a este cambio es que todos los objetos que tengan una referencia a nuestro vendedor promovido deberán enterarse de este cambio (y seguramente no lo hagan) para referenciar al nuevo objeto que lo reemplaza. La consecuencia de esto es o bien una complejidad espantosa para mantener las referencias o un sistema inconsistente.

¿Cómo se soluciona este problema? Si cambiamos el modelo para que las categorías sean un objeto aparte que la clase conozca y delegamos en este objeto todo aquello que corresponda, solucionamos ambos problemas a la vez, ya que el valor de las referencias sí puede ser cambiado en tiempo de ejecución, es sólo settear un atributo. La composición introduce menos dependencias que la herencia, y hace menos suposiciones sobre cómo se usará el programa en el futuro.

Herencia	Composición
✸ Estática (no puedo cambiar la clase en run-time, si creo otro objeto pierdo la identidad) ✸ Menos objetos, Menos complejidad ✸ ¡Es una relación entre clases! Es mucho más fuerte que la relación entre instancias planteada en composición. La herencia implica no sólo heredar código sino conceptos.	✸ Dinámico (la implementación se puede cambiar en run-time, se basa sólo en un atributo que se puede settear con otro objeto que sea polimórfico) ✸ Aumenta la cantidad de objetos → Mayor **complejidad** (Es más complicado de entender y hay que configurar adecuadamente las relaciones entre los objetos), pero más **flexibilidad** ✸ Se reparten mejor las responsabilidades en objetos más chicos.

Manejo de Errores

Asumimos que la ejecución de un programa salió bien a menos de que algo nos diga lo contrario. Si ese es el caso, queremos que mi mundo siga operando a pesar de ese error (problemas de diversa índole que hacen que el sistema se comporte de una forma no esperada), y no quede en un estado inconsistente. Por ejemplo, si una impresora no pudo imprimir un documento porque un cartucho estaba vacío, no queremos que lo demás deje de funcionar, sino que se lance un error. Si tratara de condicionar a que ese error no pase (validando si está vacío), lo que puede ocurrir es que no se cumpla con la operación esperada, pero no lance ningún error (puedo hacerlo si el dominio lo permite): No me entero de que algo no salió bien, puedo ejecutar cosas que no deberían ejecutarse si la operación falla, me es más difícil ver en qué me estoy equivocando/teniendo un error. También podría tener un mensaje de consulta y en base a si puede imprimir ejecutar el resto (mensajes armónicos, uno necesita del otro), pero una persona puede obviar esa consulta o mis objetos pueden cambiar su estado inmediatamente después de la pregunta. Otra opción sería que cada método devuelva true o false, pero esto hace que el usuario deba interpretar el error, que se repita mucho código y lógica, que sea difícil saber de dónde proviene el error, y que se comprometan todos los objetos involucrados cuando alguno de ellos falle. "Todo método debe hacer lo que promete, o no hacer nada y explotar"

Lanzado de Excepciones

Una excepción es la indicación de un problema que ocurre durante la ejecución de un programa. La principal particularidad de las excepciones es que cortan el flujo de ejecución hasta que alguien se encargue de resolverlo. Detenemos la ejecución de la sentencia que estamos evaluando en **todos sus niveles** (impresora, cabezal y cartucho). Lo hacemos con la palabra "throw" e instanciando a la clase Exception: throw new Exception(message="no hay impresoras")

Atrapar Errores

Una vez que se produce el error debemos tener una forma de recuperarnos del mismo para que el programa no termine con excepción. Donde sabemos qué hacer ante un problema, atrapamos la excepción que causó el problema y evaluamos un determinado código para seguir adelante correctamente. Para eso primero tenemos que saber qué parte del código a ejecutar es el que podría terminar en excepción, luego qué tipo de error queremos tratar y finalmente qué se debería hacer al respecto: Usamos try y catch. Cuando hay un error, el try corta la ejecución y pasa a ejecutar el catch. Si pasa algo en el catch, podemos modelar otra excepción

```
try{-código-que-quiero-ejecutar}catch error{-código-a-ejecutar-en-caso-de-error}
```

De todas formas, esa variante del catch (error) hace que se use por cualquier error. Podemos entonces modelar un catch que se active para ciertos errores. Definiendo una clase para el error: class SinCargaException inherits DomainException{const property carga}, que luego instanciamos en el trhow del cartucho: trhow new SinCargaException{...}, y capturándola en el catch: catch error: SinCargaException{...}, y así solo captura las excepciones de ese tipo. Podemos usar la cantidad de catchs que queramos, siempre definiendo uno nuevo: catch e: ...{...}catch error{...}

SOLO ATRAPAMOS UN ERROR CUANDO TENEMOS UNA LÓGICA PARA RESOLVERLO

Cuando quiero ejecutar algo independientemente de si mi programa falla o no, uso "then always"

Diagrama de Clases y de Objetos

```
object emilia{
   method entrena(ave){
          ave.comer(5)
          ave.volar(10)
          ave.comer(5)
}}
```

Diagrama estático de Objetos

Snapshot o foto de sistema en un momento determinado. Representamos a los objetos en un rectángulo, con sus atributos e interfaz separados por líneas horizontales. Representamos que un objeto "conoce a otro", señalando con una flecha en dirección al objeto conocido. También puede ser que los conozcan desde fuera del ambiente (flecha sin punto de partida que apunta a un objeto), que dos objetos se conozcan entre ellos (dos flechas), o que un objeto conozca a varios (salen muchas flechas de él).

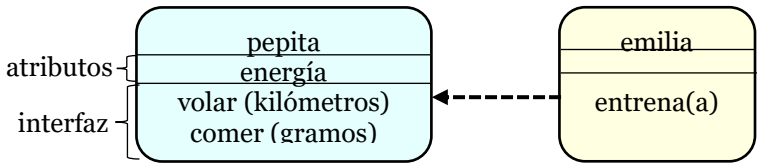


Diagrama de clases

Herramienta que permite modelar las relaciones entre las entidades: Se representan por un rectángulo que posee tres divisiones: Nombre, atributos y operaciones; aunque podemos representarlas únicamente con el nombre y la interfaz. Las clases se relacionan entre ellas. Las interfaces se representan con un recuadro punteado, con su nombre y métodos.

Relaciones entre clases

Relación "HEREDA": Se representa con una flecha de punta blanca que apunta hacia la superclase. Cuando una subclase redefine el comportamiento de su superclase, se escribe el nombre de ese método en la nueva clase, sino no.

Relación "IMPLEMENTA": Se representa con una flecha de punta blanca punteada. Se establece entre una clase y una interfaz; esto implica que la clase debe implementar todas las operaciones que defina la interfaz.

Relaciones entre Objetos

Relación "USA": Dependencia: uno de los elementos usa o depende del otro cuando: El objeto de clase A recibe o devuelve como parámetro de alguno de sus métodos un objeto de clase B, o si el objeto de clase A instancia un objeto de clase B (pero no lo almacena como variable de instancia, sólo vive como variable local en el contexto de un método). Este tipo de relación indica que los dos elementos colaboran entre sí, pero que esa relación es débil, casual; tiene un contexto temporal que no trasciende más allá de una operación. No obstante, sabemos que los cambios en la clase B podrían impactar en alguna medida en la clase A. Se representa con una flecha punteada.

Relación "CONOCE": Asociación: Uno de los elementos conoce al otro, almacenándolo como variable de instancia. Puede definirse una etiqueta que expresa el rol que cumple dicha relación. En las asociaciones, hay una relación más fuerte que en las dependencias (uso) entre ambos elementos. El conocimiento implica que la colaboración excede el marco temporal de una operación, aunque cada uno de los objetos sigue teniendo objetivos diferentes. Se representa con una flecha en dirección a quién conoce.

Ley de Demeter para Modelar

- You can play with yourself. → podemos enviarnos mensajes a nosotros mismos
- You can play with your own toys (but you can't take them apart) → Puedo mandarles mensajes a mis atributos, Pero no está bueno meterme con su composición interna
- You can play with toys that were given to you→ puedo hablar con los parámetros que lleguen
- And you can play with toys you've made yourself. → puedo hablar con los objetos que yo haya creado