PARADIGMAS DE PROGRAMACIÓN

ÍNDICE

PRESENTACIONES

Esta unidad introductoria, mucho no te puede llegar a servir, pero es buena guía para destacar los puntos fuertes o débiles de cada paradigma.

UNIDAD I — INTRODUCCIÓN

Un paradigma está constituido por los supuestos teóricos generales, las leyes y las técnicas para su aplicación que adoptan los miembros de una determinada comunidad científica.

Es un modelo o esquema fundamental que organiza nuestras opiniones con respecto a algún tema en particular.

El instrumental y las técnicas instrumentales necesarios para hacer que las leyes del paradigma se refieran al mundo real. La aplicación en astronomía del paradigma newtoniano requiere el uso de diversos telescopios, junto con técnicas para su utilización y diversas técnicas para corregir los datos recopilados.

Los paradigmas establecen límites, para resolver problemas dentro de estos, y de esta forma mejorar o proporcionar nuevas soluciones, ya que estos filtran experiencias, se ajustan a los límites, percepciones o creencias, lo que se denomina efecto paradigma.

Representan un enfoque particular o filosofía para la construcción del software. No es mejor uno que otro sino que cada uno tiene ventajas y desventajas. También hay situaciones donde un paradigma resulta más apropiado que otro.

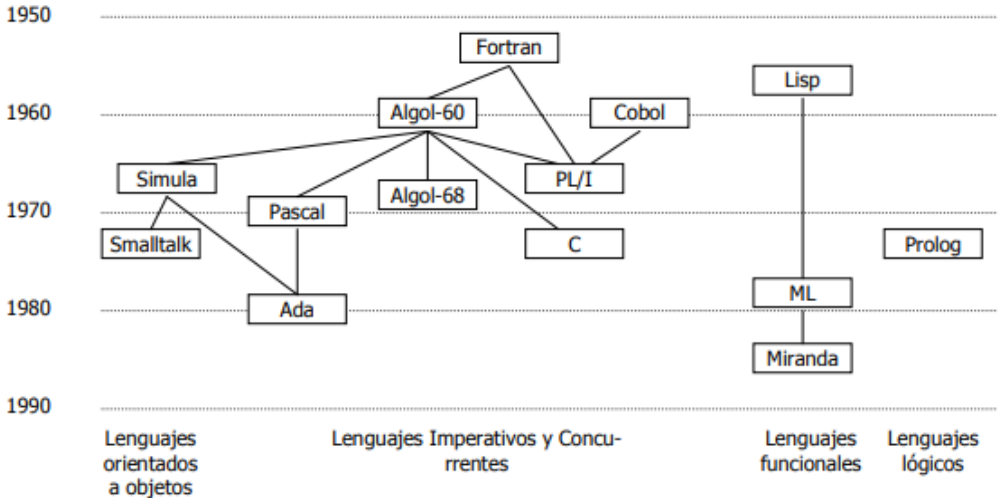
Paradigma	Bloques de Construcción	Relaciones entre los bloques de construcción
Funcional	Funciones	Composición
Procedural	Procedimientos o Secuencias de Comandos	Jerarquía
Orientado a Objetos	Clases	Herencia

Programming paradigms are heuristics used for algorithmic problem solving. A programming paradigm formulates a solution for a given problem by breaking the solution down to specific building blocks and defining the relationship among them.

Un poco de Historia

Los lenguajes de programación de nuestros días fueron desarrollados a principios de los años 50.Numerosos conceptos fueron inventados, probados e incorporados a distintos lenguajes. Con algunas pocas excepciones, el diseño de cada lenguaje tuvo una fuerte influencia de experiencias tomadas de lenguajes anteriores.

El gráfico siguiente nos muestra una cronología de cómo fueron desarrollándose los lenguajes actuales y nos demuestra que los mismos no son un producto terminado sino que nuevos y revolucionarios conceptos y paradigmas están siendo desarrollados.



Los primeros lenguajes de alto nivel fueron el Fortran y el COBOL. Fortran introdujo las expresiones simbólicas y subprogramas con parámetros y COBOL introdujo el concepto de descripción de datos. El primer lenguaje diseñado para algoritmos de comunicación más que sólo para la programación de computadoras fue el Algol-60. Introdujo el concepto de estructura de bloques, variables, procedimientos, etc. que podían se declarados en cualquier lugar del programa donde fueran necesarios.

Influenció a numerosos lenguajes posteriores tan fuertemente que fueron llamados lenguajes estilo Algol.

Fue el Pascal, el lenguajes estilo Algol que se hizo más popular porque es simple, sistemático e implementable eficientemente. Pascal y Algol-68 fueron los primeros lenguajes con una rica estructura de control, ricos tipos de datos y definiciones de tipos.

Un sucesor poderoso del Pascal: el Ada, introdujo los conceptos de paquetes y genéricos, diseñados para ayudar a la construcción de grandes programas modulares. Ada también fue un intento de sus diseñadores de construir el lenguaje de propósito general estándar.

Ciertas tendencias pueden discerniese en la historia de los lenguajes de programación. Una de las tendencias es lograr cada vez un nivel de abstracción más alto.

Los nemónicos y etiquetas del lenguaje ensamblador es una abstracción de los códigos de operación y dirección de máquina.

- Las variables y la asignación es una abstracción del almacenar o recuperar un dato de un almacenamiento de memoria.
- Las estructuras de datos son una abstracción de las estructuras de almacenamiento.
- Las estructuras de control son una abstracción de los saltos (jumps).
- Los procedimientos son una abstracción de las subrutinas.
- Las estructuras de bloques y módulos son una forma de encapsulación que ayuda a desarrollar programas modulares.

Otra tendencia fue la proliferación de paradigmas de programación. Todos los lenguajes mencionados son lenguajes de programación imperativos caracterizados por el uso de comandos que actualizan variables.

El paradigma de programación imperativo es aún el dominante, aunque otros paradigmas están rápidamente ganando popularidad.

Lisp en su forma pura, está basado enteramente en funciones sobre listas y árboles. Lisp fue el antecesor de los lenguajes de programación funcional. ML y Miranda son lenguajes funcionales modernos, tratan a las funciones como valores de primera clase e incorporan también un sistema de tipos avanzado.

Hasta ahora la notación matemática en su completa generalidad no está implementada. Sin embargo algunos diseñadores de lenguajes quisieron explotar un subconjunto de la notación matemática en los lenguajes de programación. Un lenguaje de programación lógica está basado en un subconjunto de la lógica matemática. La computadora es programada para inferir relaciones entre valores más que calcular valores de salida a partir de valores de entrada. Prolog hizo a la programación lógica popular y es más bien débil e ineficiente, sin embargo fue modificado para agregarle características no lógicas para hacerlo más usable como lenguaje de programación.

Paradigma Imperativo

Es procedural cuando usa procedimientos o funciones.

→ Fortran introdujo las expresiones simbólicas y subprogramas con parámetros.

→ COBOL introdujo el concepto de descripción de datos.

Es considerado el más común y está representado, por ejemplo, C, Pascal, COBOL.

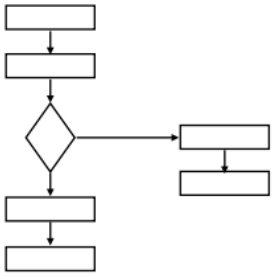
→ El **Algol-60** introdujo el concepto de estructura de bloques, variables, procedimientos, etc. Este Influenció a numerosos lenguajes posteriores tan fuertemente que fueron llamados lenguajes estilo Algol.

→ **Pascal** fue **el lenguaje estilo Algol** que se hizo más popular porque es simple, sistemático e implementado eficientemente.

Describe la programación en términos del estado del programa y sentencias que cambian dicho estado. Los programas imperativos son un conjunto de instrucciones que le indican al computador cómo realizar una tarea. **Tiene estrecha relación para modelar el mundo real**. Los programas se escriben para modelar procesos del mundo real que afectan a objetos del mundo real, y tales objetos a menudo poseen un estado que varía con el tiempo. Las variables modelan tales objetos y los programas imperativos tales procesos.

Las recetas y las listas de revisión de procesos, a pesar de no ser programas de computadora, son también conceptos familiares similares en estilo a la programación imperativa; cada paso es una **instrucción**.

La implementación de hardware de la mayoría de computadores es imperativa; prácticamente todo el hardware de los computadores está diseñado para ejecutar código de máquina, que es nativo al computador, escrito en una forma imperativa. Esto se debe a que el hardware de los computadores implementa el paradigma de las Máquinas de Turing. Desde esta perspectiva de bajo nivel, el estilo del programa está definido por los contenidos de la memoria, y las sentencias son instrucciones en el lenguaje de máquina nativo del computador (por ejemplo el lenguaje ensamblador). Los lenguajes imperativos de alto nivel usan variables y sentencias más complejas, pero aún siguen el mismo paradigma.



La programación imperativa recibe este nombre porque está basada en comandos que actualizan variables contenidas en almacenamientos.

Se caracteriza esencialmente porque las instrucciones que lo componen se ejecutan secuencialmente en un orden preestablecido, sigue una **secuencia**.

La unidad base de ejecución es el programa o conjunto de instrucciones ejecutables, que se divide en una serie de **módulos o rutinas** distribuidos de acuerdo con una organización jerárquica.

En este tipo de programación los **datos** están desorganizados, son meros apéndices de los programas y tan solo proporcionan valores que sirven para realizar cálculos. En cuanto a los datos, dependiendo del lenguaje que se trate pueden ser globales, accesibles por todos los programas de la jerarquía, o locales, utilizables solo por el programa al que pertenece.

Uno de los programas situados en la raíz de la jerarquía recibe el nombre de **programa principal** y los demás se conocen con el nombre de **subprogramas, subrutinas, funciones o procedimientos**.

Dentro de este tipo de programación podemos encontrar dos divisiones:

- **Programación estructurada** o programación sin GOTO: presenta bastantes ventajas en cuanto a la facilidad de modularización, documentación y mantenimiento de los programas;
- **Programación no estructurada** que permiten realizar saltos incondicionales en la ejecución del programa mediante instrucciones GOTO.

La complejidad crece de forma exponencial => el SW es complejo! Pero también es flexible, por lo que las solicitudes de cambios pueden ser brutales, ya que pueden aparecer cambios inoportunos que pueden afectar a la arquitectura del sistema.

Paradigma Orientado a Objetos.

Es el paradigma que está tomando mayor auge en los últimos años. Ejemplo: Smalltalk

→ La programación orientada a objetos deriva de conceptos introducidos por Simula, otro lenguaje del estilo Algol.

→ Smalltalk está basado en clases de objetos, un objeto viene a ser como una variable que puede accederse sólo a través de operaciones asociadas con él.

La Programación Orientada a Objetos (POO u OOP según siglas en inglés) es una metodología de diseño de software y un paradigma de programación que define los programas en términos de "clases de objetos", objetos que son entidades que combinan estado (es decir, datos) y comportamiento (esto es, procedimientos o métodos). La **programación orientada** a objetos expresa un programa como un conjunto de estos objetos, que se comunican entre ellos para realizar tareas.

Lenguajes procedurales	Lenguajes orientados a objetos
Escriben funciones y después les pasan datos	Definen objetos con datos y métodos y después envían mensajes a los objetos diciendo que realicen esos métodos en sí mismos

Esto difiere de los lenguajes procedurales tradicionales, en los que los datos y los procedimientos están separados y sin relación. Estos métodos están pensados para hacer los programas y módulos **más fáciles de escribir, mantener y reutilizar**.

Otra manera en que esto es expresado a menudo, es que la programación orientada a objetos anima al programador a pensar en los programas principalmente en términos de tipos de datos, y en segundo lugar en las operaciones ("métodos") específicas a esos tipos de datos. Los lenguajes procedurales animan al programador a pensar sobre todo en términos de procedimientos, y en segundo lugar en los datos que esos procedimientos manejan.

Los programadores que emplean lenguajes procedurales, escriben funciones y después les pasan datos. Los programadores que emplean lenguajes orientados a objetos definen objetos con datos y métodos y después envían mensajes a los objetos diciendo que realicen esos métodos en sí mismos.

Algunas personas también diferencian la POO sin clases, la cual es llamada a veces programación basada en objetos.

Los **objetos** que son entidades que combinan estado (es decir, datos) y comportamiento (esto es, procedimientos o métodos).

Estos presentan una identidad propia, intrínseca de cada objeto la cual no se relaciona con el estado del mismo. Los estados son valores que puede tomar.

No hay dos objetos que ocupen el mismo lugar de memoria, ya que la identidad se da por el lugar que ocupa. El tener la referencia hacia el objeto nos permite poder mandarle la instrucción, por lo que podemos decir que los mensajes hacen que se disparen las operaciones que saben hacer los objetos.

Es decir, es una entidad única, la cual presenta estados y operaciones que puede tomar y realizar. No hay una relación directa entre las operaciones y los atributos del mismo y puede tener k

atributos. El estado del objeto está protegido del exterior, solo se altera con las operaciones que ofrece el mismo, por lo que un objeto sin operaciones es un objeto inútil.

Los conceptos de la programación orientada a objetos tienen origen en Simula 67, un lenguaje diseñado para hacer simulaciones, creado por Ole-Johan Dahl y Kristen Nygaard del Centro de Cómputo Noruego en Oslo. Según se informa, la historia es que trabajaban en simulaciones de naves, y fueron confundidos por la explosión combinatoria de cómo las diversas cualidades de diversas naves podían afectar unas a las otras. La idea ocurrió para agrupar los diversos tipos de naves en diversas clases de objetos, siendo responsable cada clase de objetos de definir sus propios datos y comportamiento. Fueron refinados más tarde en Smalltalk, que fue desarrollado en Simula en Xerox PARC pero diseñado para ser un sistema completamente dinámico en el cual los objetos se podrían crear y modificar "en marcha" en lugar de tener un sistema basado en programas estáticos.

La programación orientada a objetos tomó posición como la metodología de programación dominante a mediados de los años ochenta, en gran parte debido a la influencia de **C++,** una extensión del lenguaje de programación C. Su dominación fue consolidada gracias al auge de las Interfaces gráficas de usuario, para los cuales la programación orientada a objetos está particularmente bien adaptada.

Las características de orientación a objetos fueron agregadas a muchos lenguajes existentes durante ese tiempo, incluyendo Ada, BASIC, Lisp, Pascal, y otros. La adición de estas características a los lenguajes que no fueron diseñados inicialmente para ellas condujo a menudo a problemas de compatibilidad y a la capacidad de mantenimiento del código. Los lenguajes orientados a objetos "puros", por otra parte, carecían de las características de las cuales muchos programadores habían venido depender. Para saltar este obstáculo, se hicieron muchas tentativas para crear nuevos lenguajes basados en métodos orientados a objetos, pero permitiendo algunas características procedurales de maneras "seguras". El Eiffel de Bertrand Meyer fue un temprano y moderadamente acertado lenguaje con esos objetivos pero ahora ha sido esencialmente reemplazado por **Java**, en gran parte debido a la aparición de Internet, para la cual Java tiene características especiales.

Entre los lenguajes orientados a objetos destacan los siguientes:

-Smalltalk
-Objective-C C++
-Ada 95
-Java
-PowerBuilder
-Ocaml
-Python
-Delphi
-C Sharp
-Eiffel
-Ruby
-ActionScript
-Visual Basic.NET
-PHP

Paradigma Funcional (λ)

Está basado en el cálculo lambda (o λ-cálculo), de Alonso Church.

→ LISP fue el primer lenguaje basado enteramente en el Cálculo Lambda, en su forma pura, está basado *enteramente en funciones sobre listas y árboles*. Pero los lenguajes funcionales modernos, tratan a las funciones como valores de primera clase e incorporan también un sistema de tipos avanzado.

El objetivo es conseguir lenguajes expresivos y matemáticamente elegantes, en los que no sea necesario bajar al nivel de la máquina para describir el proceso llevado a cabo por el programa, y evitando el concepto de estado del cómputo. La secuencia de computaciones llevadas a cabo por el programa se regiría única y exclusivamente por la reescritura de definiciones más amplias a otras cada vez más concretas y definidas.

Los programas escritos en un lenguaje funcional están constituidos únicamente por definiciones de **funciones matemáticas,** en las que se verifican ciertas propiedades como la transparencia referencial (el significado de una expresión depende únicamente del significado de sus sub-expresiones), y por tanto, la carencia total de efectos laterales.

Otras características propias de estos lenguajes son la no existencia de asignaciones de variables y la falta de construcciones estructuradas como la secuencia o la iteración (lo que obliga en la práctica a que todas las repeticiones de instrucciones se lleven a cabo por medio de *funciones recursivas*).

Existen dos grandes categorías de lenguajes funcionales: los funcionales puros y los híbridos. La diferencia entre ambos estriba en que los lenguajes funcionales híbridos son menos dogmáticos que los puros, al admitir conceptos tomados de los lenguajes procedimentales, como las secuencias de instrucciones o la asignación de variables. En contraste, los lenguajes funcionales puros tienen una mayor potencia expresiva, conservando a la vez su transparencia referencial, algo que no se cumple siempre con un lenguaje funcional híbrido.

Entre los lenguajes funcionales puros:

-Haskell

-Miranda

Los lenguajes funcionales híbridos:

-Lisp

-Scheme

-Ocaml

-Standard ML (estos dos últimos, descendientes del lenguaje ML).

Paradigma Lógico

→ Un ejemplo es PROLOG. PROLOG hizo a la programación lógica popular y es más bien débil e ineficiente, sin embargo fue modificado para agregarle características no lógicas para hacerlo más usable como lenguaje de programación.

Un lenguaje de programación lógica está basado en un subconjunto de la lógica matemática. La computadora es programada para inferir relaciones entre valores más que calcular valores de salida a partir de valores de entrada.

La mayoría de estos lenguajes se basan en la lógica de predicados. Se definen propiedades de los objetos del dominio de referencia y relaciones entre ellos a través de fórmulas, las cuales usan normalmente el conector implicación.

Paradigma Orientado a Eventos

La programación dirigida por eventos es un paradigma de programación en el que tanto la estructura como la ejecución de los programas van determinados por los sucesos que ocurran en el sistema o que ellos mismos provoquen.

Será el propio usuario -o lo que sea que esté accionando el programa- el que dirija el flujo del mismo.

En la programación dirigida por eventos, al comenzar la ejecución del programa se llevarán a cabo las inicializaciones y demás código inicial y a continuación el programa quedará bloqueado hasta que se produzca algún evento. Cuando alguno de estos eventos tenga lugar, el programa pasará a ejecutar el código del correspondiente manejador de evento

- Clic del mouse.
- Llegada de un mensaje por la red.

Programación Orientada a Aspectos

La Programación Orientada a Aspectos (POA) es un paradigma de programación relativamente reciente cuya intención es permitir una adecuada modularización de las aplicaciones y posibilitar una mejor separación de conceptos. Gracias a la POA se pueden capturar los diferentes conceptos que componen una aplicación en entidades bien definidas, de manera apropiada en cada uno de los casos y eliminado las dependencias inherentes entre cada uno de los módulos. De esta forma se consigue razonar mejor sobre los conceptos, se elimina la dispersión del código y las implementaciones resultan más comprensibles, adaptables y reusables. Varias tecnologías con nombres diferentes se encaminan a la consecución de los mismos objetivos y así, el término POA es usado para referirse a varias tecnologías relacionadas -como los métodos adaptativos, los filtros de composición, la programación orientada a sujetos o la separación multidimensional de competencias.

El hecho es que hay ciertas decisiones de diseño que son difíciles de capturar con las técnicas antes citadas, debiéndose al hecho de que ciertos problemas no se dejan encapsular de igual forma que los que habitualmente se han resuelto con funciones u objetos. La resolución de éstos supone o bien la utilización de repetidas líneas de código por diferentes componentes del sistema, o bien la superposición dentro de un componente de funcionalidades dispares.

La programación orientada a aspectos, permite, de una manera comprensible y clara, definir nuestras aplicaciones considerando estos problemas. Por aspectos se entiende dichos problemas que afectan a la aplicación de manera horizontal y que la programación orientada a aspectos persigue poder tenerlos de manera aislada de forma adecuada y comprensible, dándonos la posibilidad de poder construir el sistema componiéndolos junto con el resto de componentes.

Entre los **objetivos** que se ha propuesto la POA están, principalmente, el de separar conceptos y el de minimizar las dependencias entre ellos. Con el *primer objetivo* se persigue que cada decisión se tome en un lugar concreto y no diseminado por la aplicación. *Con el segundo*, se pretende desacoplar los distintos elementos que intervienen en un programa. Su consecución implicaría las siguientes ventajas:

- Un código menos enmarañado, más natural y más reducido.
- Mayor facilidad para razonar sobre los conceptos, ya que están separados y las dependencias entre ellos son mínimas.
- Un código más fácil de depurar y más fácil de mantener.
- Se consigue que un conjunto grande de modificaciones en la definición de una materia tenga un impacto mínimo en las otras.
- Se tiene un código más reusable y que se puede acoplar y desacoplar cuando sea necesario.

UNIDAD II — CONCEPTOS COMPARADOS EN LENGUAJES DE PROGRAMACIÓN

La programación imperativa, al igual que los demás tipos de programación, posee elementos y características que son importante destacarlos, como por ejemplo valores y tipos, expresiones, variables, comandos, enlaces etc.

Los **algoritmos** pueden ser representados de diferentes maneras, con diferentes estructuras.

Valores y Tipos

Saber diferenciar tipos y valores. Definir los diferentes tipos y en especial explicar el compuesto, definiendo cada uno su forma y su cardinalidad, relacionandolo con ejemplos de su uso. Casi no pregunta.

Llamaremos **valor** a algo que pueda ser evaluado, almacenado, incorporado a una estructura de datos, pasado como argumento a un procedimiento o función, retornado como resultado de una función etc. En otras palabras, un valor es cualquier entidad que existe durante una computación.

Si el valor es atómico, no se puede dividir; en cambio una colección de edades por ejemplo si se pueden sub-dividir, porque son compuestos (está compuesto por valores más simples, un arreglo de registros por ejemplo).

Por ejemplo en **Pascal** encontramos los siguientes valores:

- Valores primitivos (valores de verdad, caracteres, enumerados, enteros y números reales)
- Valores compuestos (registros, arreglos, conjuntos y archivos)
- Punteros
- Referencias a variables
- Abstracción de procedimiento y función.

Los tres primeros grupos son evidentes que son valores, sin embargo los otros dos también son considerados como valores porque pueden ser pasados como argumentos. Englobaremos con el término abstracción a las funciones y procedimientos y a cualquier otra entidad similar en otro lenguaje.

Los **valores** pueden ser agrupados en tipos, como sugiere la lista anterior. Como ejemplo podemos decir que siempre se hace una distinción entre valores de verdad y los números enteros, debido a que en los enteros se pueden hacer operaciones de suma y multiplicación que no se pueden hacer con los valores booleanos.

¿Qué es exactamente un tipo? La respuesta más obvia, es quizás, que un tipo es un conjunto de valores. Cuando decimos que v es un valor del tipo T, significa simplemente que $v \in T$. Cuando decimos que una expresión E es del tipo T, significa que el resultado de evaluar la expresión E dará un valor de tipo T. Es decir, es todo *conjunto de valores con las mismas características y que se comportan de la misma manera*.

Sin embargo, **no todos los conjuntos de valores pueden decirse que pertenecen a un mismo tipo.** Debido a que todos los valores de un tipo deben exhibir un **comportamiento uniforme** bajo operaciones asociadas a ese tipo. Por ejemplo el conjunto {23, verdadero, Lunes} no es un tipo; pero {verdadero, falso} si lo es porque exhibe un comportamiento uniforme bajo las operaciones lógicas de negación, conjunción y disyunción; y {..., -2, -1, 0, 1, 2, ...} también lo es, porque todos sus valores exhiben un comportamiento uniforme bajo las operaciones de suma, multiplicación, etc.

Todos los lenguajes de programación poseen tanto tipos primitivos, los cuales son valores atómicos, como tipos compuestos cuyos valores están compuestos por valores simples. Algunos lenguajes tienen también tipos recursivos, los cuales están compuestos por otros valores del mismo tipo.

Tipos primitivos o simples

Los tipos primitivos son aquellos que están compuestos por valores atómicos y por lo tanto no pueden ser descompuestos en valores más simples. Estos tipos pueden variar con cada lenguaje dependiendo de la especialización del mismo, ya sea comercial (COBOL) o científico (Fortran) etc.

Seguiremos la siguiente notación para los tipos primitivos:

Valor_de_verdad	`{verdadero, falso}`
Entero	`{..., -2, -1, 0, 1, 2, ...}`
Real o flotante	`{..., -1.0, ..., 0.0, ..., 1.0, ...}`
Carácter	`{..., 'a', 'b', ..., 'z', ...}`

No todos los leguajes coinciden con esas definiciones de tipos, pero esos son los generales. En Pascal y Ada se puede definir un tipo primitivo completamente nuevo enumerando sus valores (más precisamente, enumerando identificadores que denotarán sus valores). Este tipo de datos se llama **tipo enumerado**.

Meses `{ene, feb, mar, abr, may, jun, jul, ago, sep, oct, nov,dic}`
Data Color = `Rojo | Verde | Azul`

Dentro de ese conjunto hay etiquetas, las cuales son valores atómicos.

En algunos lenguajes se pueden definir subconjuntos de un tipo existente, obteniendo un tipo **subrango**. Por ejemplo el tipo subrango 28..31 tiene los valores {28, 29, 30, 31} y es un subconjunto de enteros, el cual debe estar compuesto por un conjunto de valores consecutivos. Obliga a tener en cuenta que solo se pueden definir sub-rangos aquellos valores que tengan un orden (caracteres, enteros, etc).

Un punto interesante es la **cardinalidad de un conjunto (o de un tipo).** Representa la cantidad de valores del tipo. Escribiremos **#S** para significar el número de valores distintos en S.

Por ejemplo:

#valor_de_verdad = 2
#meses = 12
#enteros = `2 x maxint + 1 (en Pascal)`

Tipos compuestos

Un tipo compuesto (o tipo de datos estructurado) es un tipo cuyos valores están compuestos o estructurados a partir de valores más simples. Los lenguajes de programación soportan una amplia variedad de estructuras de datos: tuplas, registros, arreglos, conjuntos, strings, listas, arboles, archivos secuenciales, archivos directos, etc.

Dependiendo de cómo se combinen estos, pueden hacerse aún más complejos. Todos estos tipos pueden ser estudiados a partir de un pequeño número de conceptos:

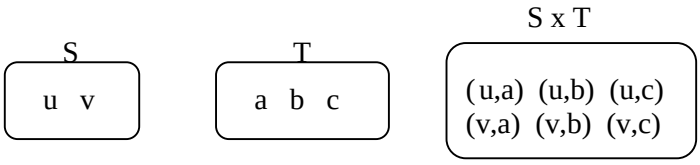
- Producto cartesiano (tuplas y registros)
- Uniones disjuntas (variants y uniones)
- Mapeo (arreglos y funciones)
- Conjunto Potencia (conjuntos)
- Tipos recursivos (estructuras de datos dinámicas)

Cada lenguaje de programación tiene su propia notación para definir tipos compuestos. Introduciremos una notación matemática simple y estándar.

Producto Cartesiano

Un tipo simple de composición de valores es el Producto Cartesiano, donde los valores de dos tipos (posiblemente diferente) son apareados. S x T es el conjunto de todos los pares ordenados de valores, donde el primer valor de cada par se toma del conjunto S y el segundo del T. Formalmente:

$$S \times T = \{ (x,y) / x \in S; y \in T\}$$



La cardinalidad del producto cartesiano es #(S x T) = #S x #T. Entendiéndose por cardinalidad de un tipo a la cantidad de valores del mismo.

Se puede extender la idea de pares ordenados a tripletes, cuadrupes etc. Podemos tomar como ejemplo de producto cartesiano la definición de un **registro:**

```
Type    Fecha: record
        m :meses;
        d :1..31;
        end;
```

El tipo `Fecha` tiene el siguiente conjunto de valores {ene, feb, ..., dic} x {1, 2, 3, ..., 31}. O sea 372 pares de la forma (ene, 1) (ene, 2) ... (feb, 1) ... (dic, 31). La cardinalidad está dada #S x #T=>12 x 31=372 *Es la cantidad de combinaciones posibles que pueden aparecer.*

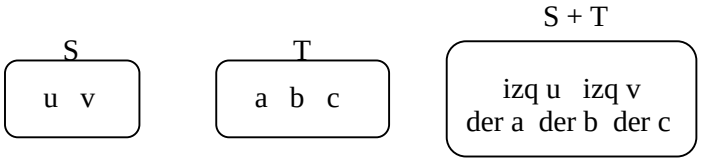
Un **caso especial de producto cartesiano** es donde todos los componentes de las tuplas son tomados del mismo conjunto S. Las tuplas se dicen homogéneas. Escribiremos: $\mathbf{S^n = S \times S \times ... \times S}$. En este caso la cardinalidad estará dada por $\#(S^n) = \#(S)^n$

Finalmente consideremos un caso especial donde n = 0. La ecuación anterior nos dice que S^0 tendrá un único valor => $\#S^n=1$. Este valor es la 0-tupla(). Una tupla sin ningún componente. Este tipo será útil para definir un tipo con un único valor: Unit = { () } Se corresponde con el unit de ML y **void (vacío**, re presenta la ausencia de valor con un valor) de algol-68 y C. Unit no es un conjunto vacío, contiene una única tupla que no contiene ningún componente.

Uniones Disjuntas

Otro tipo de composición de valores son las uniones disjuntas donde los valores se toman de uno de dos tipos (normalmente diferentes). Lo que hace es juntar los elementos de un tipo S y de un Tipo T y los pone todos adentro de una misma "bolsa". S + T es el conjunto de valores donde cada valor es tomado ya sea de S o de T y es **rotulado (izq o der) para indicar de qué conjunto fue tomado**:

$$S + T = \{izq\ x / x \in S\} \cup \{der\ y / y \in T\}$$



La cardinalidad está dada por #(S + T) = #S + #T Podemos extender esta operación a n tipos: S1, S2... Sn en donde cada uno de estos puede ser rotulado por i.

```
type
    TipoDato = (Num, Fech, Str);
    Fecha = record D, M, A: Byte; end;

    Ficha = record
        Nombre: string[20];              (* Campo fijo *)
        case Tipo: TipoDato of           (* Campos variantes *)
                Num: (N: real);          (* Si es un número: campo N *)
                Fech: (F: Fecha);        (* Si es fecha: campo F *)
                Str: (S: string[30]));   (* Si es string: campo S *)
        end;
var
    UnDato: Ficha;

UnDato.Nombre  := 'Nacho';
UnDato.Tipo    := Num;
UnDato.N       := 3.5;
Writeln('Ahora el tipo es numérico, el nombre es ',  UnDato.Nombre
Writeln('Campo N: ', UnDato.N);
```

Supongamos que se desea almacenar datos de personas en las que se contempla el Estado Civil, para el cual se guardará distinta información si es soltera, casada, divorciada o viuda.


```
type
EstadoCivil = (ecSoltero, ecCasado, ecViudo, ecDivorciado);
Fecha = record D, M, A: byte; end;

Persona = record
    ApyNom: string[20];          (* Campos fijos *)
    Edad: integer;
    case Estado: EstadoCivil of (* Campos variantes *)
        ecSoltero: ();
        ecCasado: (conyuge: string[20]; FBoda: Fecha);
        ecViudo, ecDovorciado : (FFin: Fecha);
    end;
```

```
var
  V1, V2, V3, V4, V5: Variant;
  I: Integer;
  D: Double;
  S: string;

begin
  V1 := 1;              { integer value }
  V2 := 1234.5678;      { real value }
  V3 := 'Hello world!'; { string value }
  V4 := '1000';         { string value }
  V5 := V1 + V2 + V4;   { real value 2235.5678}
  I := V1;              { I = 1 (integer value) }
  D := V2;              { D = 1234.5678 (real value) }
  S := V3;              { S = 'Hello world!' (string value) }
  I := V4;              { I = 1000 (integer value) }
  S := V5;              { S = '2235.5678' (string value) }
end;
```

VARIANT es la “bolsa”!!

Se pueden definir tipos compuestos que tengan círculos y rectángulos como valores constitutivos:

```
data Forma = Circulo Float | Rectangulo Float Float

area :: Forma -> Float
area (Circulo r) = pi * r * r
area (Rectangulo base altura) = base * altura
```

Ejemplo en Haskell. El Tipo es FORMA => con el | une círculos y rectángulos => Se define la función área que en su definición recibe como parámetro de entrada una forma (circulo o rectángulo) y devuelve un flotante => la función se define por tramos.

Cabe aclarar que **la unión disjunta no es lo mismo que la unión común**. El rótulo nos permite identificar de dónde un valor fue tomado. Por ejemplo si T = {a, b, c}

T ∪ T = {a, b, c} = T

T + T = {izq a, izq b, izq c, der a, der b, der c}

Mapeo

"Asociación" entre tipos

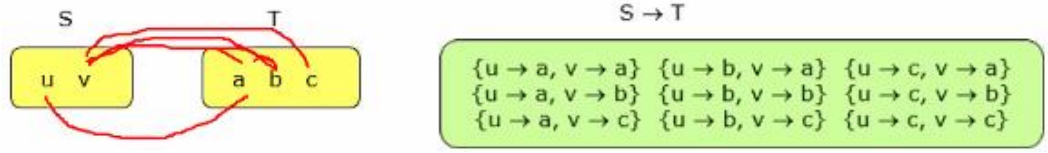
La noción de mapeo o función de un conjunto a otro es muy importante en los lenguajes de programación. Consideremos una función m que relaciona cada valor x en S con un valor de T. Los valores de T son llamados imágenes de x bajo m y se escribe m(x).

m: S → T es una función de S en T.

Si consideramos a S = {u, v} y T = {a, b, c}. Por ejemplo m(x) = {u → a, v → c} puede ser una función definida de S en T. Donde S debe tener un orden

Con la notación S → T definimos el conjunto de todas las funciones posibles de S en T.

$$S \to T = \{m / x \in S \Rightarrow m(x) \in T\}$$



La **cardinalidad** está dada por $\#S \to T = (\#T)^{\#S}$

Por ejemplo **un array puede ser entendido como un mapeo**:

```
Array [S] of T

Type color = (rojo, verde, azul);
     Pixel =  array [color] of 0..1
```

El conjunto de valores del tipo arreglo es pixel = color → {0, 1} donde color = {rojo, verde, azul} Los valores para el tipo pixel está dado por ocho combinaciones.

{rojo → 0, verde → 0, azul → 0}
{rojo → 0, verde → 0, azul → 1}
{rojo → 0, verde → 1, azul → 0}
{rojo → 0, verde → 1, azul → 1}
{rojo → 1, verde → 0, azul → 0}
{rojo → 1, verde → 0, azul → 1}
{rojo → 1, verde → 1, azul → 0}
{rojo → 1, verde → 1, azul → 1}

La mayoría de los lenguajes de programación permiten definir arreglos multidimensionales. Podemos pensar en un arreglo multidimensional como tener un único índice que sea una n-tupla.

Por ejemplo:

```
Type ventana = array [0..799, 0..599] of 0..1
```

El conjunto de valores de este tipo es ventana = {0, 1, …, 799} x {0, 1, …, 599} → {0, 1} y está indexado por un par ordenado de enteros.

Además de la forma de arreglos, los mapeos ocurren en los lenguajes de programación en la forma de abstracción de función. Una función implementa un mapeo de S → T por el significado del algoritmo que toma cualquier valor de S (el argumento) y calcula su imagen en T (el resultado). El conjunto S no necesita ser finito. Una función SIEMPRE es un mapeo.

Consideremos otro la siguiente función definida en Pascal:

```
Function par (n: Integer) : Boolean;
      Begin
        Par := (n mod 2 = 0);
      End;
```

Esta función implementa un mapeo particular definido de Integer → boolean (*de enteros en booleanos*) y sus valores son: {0 → true, ±1 → false, ±2 → true, ±3 → false,...}

Una función `impar`, implementará otro mapeo de Integer → boolean.

Si una función recibe n valores como parámetros podemos interpretarlo como un simple argumento. Una **n-tupla.**

Conjunto Potencia

Consideremos un conjunto de valores denominado S. Estamos interesados en valores que son entre sí, subconjuntos de S. El conjunto de todos los subconjuntos se llama Conjunto Potencia de S y se escribe formalmente: $\wp S = \{s \mid s \subseteq S\}$

Las operaciones básicas a un conjunto son las operaciones usuales de la teoría de conjuntos: pertenencia, inclusión, unión e intersección.

Cada valor de S puede ser miembro o no de un conjunto particular en $\wp S$. Y además la cardinalidad del conjunto potencia de S está dada por: $\#(\wp S) = 2^{\#S}$

Por ejemplo, considere la siguiente definición Pascal...

```
Type color = (rojo, verde, azul);
      ConjPot = set of color
```

El conjunto de valores de este tipo es PoSet = $\wp$color es el conjunto de todos los subconjuntos de Color = {rojo, verde, azul}. Los siguientes 8: { } {rojo} {verde} {azul} {rojo, verde} {rojo, azul} {verde, azul} {rojo, verde, azul}

Pascal permite solo conjuntos de valores primitivos (boolean, char, enumeados y enteros).

Tipos Recursivos

RECURSIVIDAD => Se define en sí mismo. Ejemplo: fractales.

Se considera un tipo recursivo a aquel que se compone de valores del mismo tipo y se define en términos de sí mismo.

Uno de los tipos recursivos son las listas las cuales pueden tener cualquier número de componentes inclusive ninguno.

Supóngase que queremos definir un tipo cuyos valores son listas de enteros. Podemos definir dicha lista de enteros como un valor que puede ser vacío o un par que consiste de un entero (cabeza) y la una lista de enteros (cola). Esta definición es recursiva y puede ser escrita como:

Lista_enteros = nil unit + cons (Entero x **Lista_enteros**) Elemento nulo o bien un entero concatenado con una lista de enteros. En otras palabras:

Lista_enteros = { nil () } ∪ {cons (i,l) / i ∈ Entero, l ∈ **Lista_enteros**} Donde nil y cons indicarían los rótulos considerados en el producto cartesiano.

En definitivas esta definición puede ser descripta por los siguientes conjuntos:

{ nil }
∪ {cons (i, nil) / i ∈ Entero }
∪ {cons (i, cons(j, nil)) / i, j ∈ Entero }
∪ {cons (i, cons(j, cons(k, nil))) / i, j, k ∈ Entero }
∪ ...

data [a] = [] | a : [a] en Haskell

Generalizando la ecuación de un conjunto recursivo:

L = Unit + (S x L) que es un conjunto todas las listas finitas de valores tomados de S.

En general el conjunto de valores de un tipo T, se definirán por una ecuación recursiva de la forma: T =... T ...

La solución de la ecuación puede determinarse de la siguiente manera: reemplazar el conjunto vacío para T en la parte derecha de la ecuación, esto da una primera aproximación de T. Luego sustituir la primera aproximación de T en la parte derecha de la ecuación nuevamente para obtener la segunda aproximación de T y así sucesivamente.

La cardinalidad de un tipo recursivo es infinito. Por lo tanto nunca podremos enumerar todos los valores para un tipo recursivo.

Otro tipo recursivo son los árboles; la definición de un árbol en ML puede ser:

```
datatype
      arbol = hoja of int
      | rama of arbol * arbol
```

Según la definición de producto cartesiano e unión disjunta será: arbol = int + (arbol x arbol)

Control de Tipos

Tema fundamental estudiarlo, saber explicar para que se hace control de tipos, los diferentes tipos y hacer comparaciones de ellos. No está muy completo la comparación en cuanto a ventajas y desventajas, tratar de tomar la de apuntes

Agrupar valores en tipos permite al programador **describir los datos**. **Previene además que los programas realicen operaciones sin sentido** como por ejemplo multiplicar un carácter por un valor de verdad. En este sentido los lenguajes de alto nivel se distinguen de los de bajo nivel donde los tipos son byte y word.

Para prevenir la realización de operaciones sin sentido la implementación de los lenguajes deben realizar una comprobación de tipos sobre los operandos. Cuando se realiza una multiplicación debe asegurarse de que los dos operandos sean números. Similarmente antes de realizar una operación lógica, el o los operandos deben chequearse para asegurarse de que sean valores de verdad.

Data types

- Data types are present in programming languages for at least three different reasons:
  - 1. At the design level, as support for the conceptual organization (effectively controllable comments);
  - 2. At the program level, as support for correctness (operaciones coherentes);
  - 3. At the translation level, as support for the implementation (acceso directo a memoria).

En qué momentos se puede realizar el control de tipos? Se puede realizar en dos momentos diferentes, los lenguajes se separan en dos grupo control de **tipo estático**: son más rígidos, cuando el programador termina de escribir todo el código y compila, ahí es cuando se hace el control. **Control de tipo dinámico** no realiza nada luego de escribir el código, lo hacen en el momento de la ejecución.

Control de Tipos Estático

- Ada, C, C++, C#, JADE, Java, Fortran, Haskell, ML, Pascal, Scala

Control de tipos Dinámico

- Groovy, JavaScript, Lisp, Lua, Objective-C, PHP, Prolog, Python, Ruby, Smalltalk, Tcl

Sin embargo hay bastante libertad en cuanto al momento de hacer el control de tipos; puede realizarse en tiempo de compilación o de ejecución; este hecho marca una importante **clasificación de los lenguajes de programación en lenguajes estáticamente tipeados y dinámicamente tipeados.**

En un **lenguaje tipeado estáticamente**, cada variable y parámetro tiene un tipo fijo y es elegido por el programador. Por lo tanto el tipo de cada expresión puede ser deducido y la comprobación de tipo se realiza en tiempo de compilación. *La mayoría de los lenguajes de alto nivel son estáticamente tipeados.*

En un **lenguaje tipeado dinámicamente**, solo los valores tienen un tipo fijo. Una variable o parámetros no tienen un tipo designado, pero pueden tomar valores de diferente tipo en diferentes momentos. Esto implica que el tipo de los operandos deben ser chequeados inmediatamente antes de realizar una operación en tiempo de ejecución. *Lisp y Smalltalk son lenguajes dinámicamente tipeados.*

Por ejemplo, considere la siguiente función Pascal **(estatico):**

```
Function par (n: integer): boolean;
 Begin
      Par := (n mod 2 = 0)
 End;
```

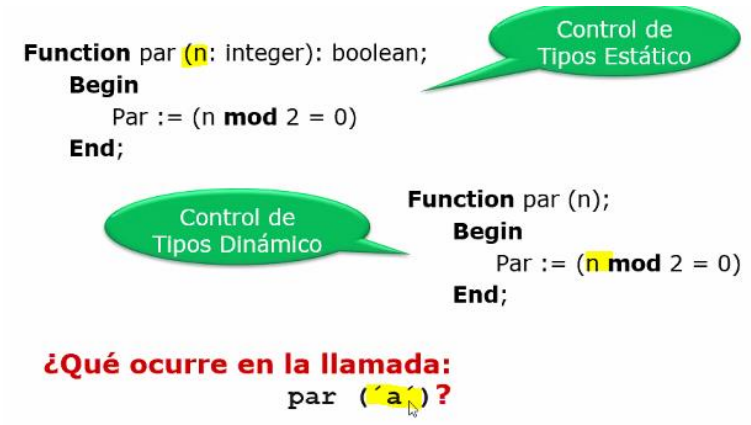
No se puede conocer de antemano el valor que tomará la variable `n`, de lo que sí podemos estar seguros es que será un valor entero debido a que fue declarado como tal. Asimismo podemos deducir que los operandos de `mod` serán ambos enteros, y que su resultado también lo será. Finalmente podemos deducir que el valor devuelto por la función será un valor de verdad como se especifica en la definición.

En cada llamada a la función par (i+1) podemos chequear que el argumento deberá ser un entero. En tiempo de compilación ya se conoce el tipo de cada variable y de cada expresión.

Considere ahora la siguiente función de un lenguaje hipotético **tipeado dinámicamente:**

```
Function par (n);
 Begin
      Par := (n mod 2 = 0)
 End;
```

El tipo de n es desconocido de antemano, por lo tanto es necesario un control de tipo en tiempo de ejecución debido a que la función mod necesita que ambos operandos sean enteros. Esta función puede ser llamada con argumentos de diferentes tipos, por ejemplo par (i+1) o par (true) o par (x) donde el tipo de x tampoco es conocido de antemano.



El **tipeo dinámico** implica que la ejecución de un programa será un poco más lenta debido al control de tipos necesario en tiempo de ejecución y también implica que cada valor debe ser rotulado con su tipo de manera de hacer posible el control de tipos.

Esta sobrecarga de tiempo y espacio son evitados en **un lenguaje estáticamente** tipeado porque todo el control de tipos se hace en tiempo de compilación. **Una ventaja importante del tipeo estático** es la seguridad de que los errores de tipos están garantizados que serán detectados por el compilador. **La ventaja de un tipeo dinámico** es su flexibilidad, la cual puede producir ciertos problemas, pero la ventaja está en que el programador se ve beneficiado a la hora de escribir el código, y que lo que escribe puede ser reutilizado sin variar el código; como ser evaluar que numero de tres es más grande => puedo reutilizar el código para saber que palabra de tres es la que está más al final del diccionario.

Equivalencia de Tipos

¿Cómo sabe el compilador cuando debe dar error o no, si los tipos coinciden o no?

No suele preguntar casi, pero con ejemplo es muy claro explicarlo.

1. Considere una operación que espera un operando del tipo T.
2. Suponga que en realidad el operando en cuestión es de tipo T'.
3. El lenguaje debe chequear si el tipo T es equivalente a T' (T ≡ T').

=>Deberíamos encontrar una forma de determinar si un tipo es equivalente a otro. Una de las posibles maneras de definir la Equivalencia de tipos podría ser:

Equivalencia Estructural:

T ≡ T' si y solo si T y T' tienen el mismo conjunto de valores, tienen la misma estructura

Mediante la equivalencia estructural se hará el chequeo de tipos comparando las estructuras de los tipos T y T'. (*No es necesario, y en general imposible, enumerar todos sus valores*).

Las siguientes reglas ilustran como podemos decidir si T ≡ T', definido en términos de productos cartesiano, uniones disjuntas y mapeos.

- Si T y T' son dos tipos primitivos.
  Luego T ≡ T' si y solo sí T y T' son idénticos.
  Ej: Integer ≡ Integer.
- Si T = A x B y T' = A' x B'.
  Luego T ≡ T' si y solo sí A ≡ A' y B ≡ B'.
  Ej: Integer x Character ≡ Integer x Character
- Si T = A + B y T' = A' + B'.
  Luego T ≡ T' si y solo sí A ≡ A' y B ≡ B' o A ≡ B' y B ≡ A'.
  Ej: Integer + Character ≡ Character + Integer
- Si T = A → B y T' = A' → B'.
  Luego T ≡ T' si y solo sí A ≡ A' y B ≡ B'.
  Ej: Integer → Character ≡ Integer → Character

- En otro caso, T no es equivalente a T'.

A pesar de que estas reglas son simples, **no es fácil ver si dos tipos recursivos** son estructuralmente equivalentes. Considere lo siguiente:

$$T = Unit\ (A \times T)$$
$$T' = Unit\ (A \times T')$$

Intuitivamente, T y T' son estructuralmente equivalente. Sin embargo, el razonamiento necesario para decidir si dos tipos recursivos arbitrarios son estructuralmente equivalentes**, hace que el chequeo de tipos sea muy difícil.**

Equivalencia de Nombres:

- T ≡ T' si y solo si T y T' fueron definidos en el mismo lugar.
- T ≡ T' si y solo si T y T' poseen los mismos nombres

Por ejemplo, considere la siguiente definición Pascal.

```
type T1 = file of Integer;
     T2 = file of Integer;

var F1: T1;
    F2: T2;

procedure p (var F: T1);
        begin  ... end;
```

La llamada a procedimiento "p (F1)" **pasaría la verificación de tipos** debido a que los tipos de F y F1 son equivalentes. Sin embargo la llamada "p (F2)" **fallaría la verificación de tipos** debido a que los tipos F y F2 no son equivalentes; T1 y T2 están definidos en diferentes declaraciones.

Entonces, ¿Cuál de los dos enfoque es mejor?

- Las razones principales por las que suele preferirse la equivalencia de nombres son:
  - La equivalencia de nombres suele ser más segura.
  - Se presume que si un programador declara dos veces el mismo tipo es porque tiene una razón para hacerlo.
    - Por lo tanto, considerar estos tipos como diferentes protege la seguridad del programa (previene al programador de hacer algo sin sentido).
  - La equivalencia de nombres es más fácil de implementar. El compilador sólo tiene que comparar las cadenas de caracteres que representan los nombres de los tipos.
  - Para implementar la equivalencia estructural, es necesario escribir una función recursiva que compare las estructuras de datos que representan los tipos de los dos objetos bajo consideración. Esto también suele impactar la eficiencia del compilador (lo hace más lento).

Si el programador decidió definir dos estructuras estructuralmente iguales pero con el mismo nombre ¡por algo lo hizo!

We say that type T is compatible with type S, if a value of type T is permitted in any context in which a value of type S would be admisible.

Principio de completitud de tipos

Las funciones y procedimientos se consideran como valores porque pueden ser pasadas como argumentos; pero no pueden ser evaluados, no pueden ser asignados, ni usados como parte de valores compuestos.

Acá hay que rescatar para qué sirve el principio, no pide que lo definas como el cuadro, si saber qué hace o quiere decir.

Por lo tanto se dice que estos valores son valores de *segunda clase*; por otro lado, los valores lógicos, enteros, registros, arreglos, etc. pueden usarse para cualquiera de estas operaciones, por lo tanto se dicen valores *de primera clase*.

```
Hugs> par 7
False
Hugs> map par [2,3,2,4,6,8]
[True,False,True,True,True,True]
Hugs> map primo [2,3,2,4,6,8]
[True,True,True,False,False,False]
Hugs> map (+3) [2,3,2,4,6,8]
[5,6,5,7,9,11]
Hugs>
```

Este tipo de distinción es común en la mayoría de los lenguajes de programación (como Fortran, Algol-60, Pascal y Ada). Sin embargo otros lenguajes (como ML, Miranda y algunas extensiones de Algol68) intentan evitar esta distinción de clases y permiten que todos los valores, incluidas las abstracciones, se manejen de manera similar.

Por ejemplo el resultado de una función en Pascal debe ser un valor primitivo o un puntero, no puede ser un string, conjunto o registro. Esta distinción no tiene lugar en lenguajes más modernos, como el Ada que permite que el resultado de una función sea de cualquier tipo.

Del otro extremo, algunos lenguajes (mencionados arriba) intentan eliminar toda distinción de clases y **de alguna manera tratan de cumplir el principio de la completitud de tipos:**

> *"Las operaciones que se puedan realizar con los valores pertenecientes a un tipo, no deberían ser arbitrariamente restrigidas al mismo".*

Si con los valores de un tipo puede tal o cual operación => esa operación puedo hacer cualquier valor. Este principio debe contribuir a que los lenguajes no posean restricciones sobre las operaciones que pueden aplicarse a determinados tipos y de esa manera reducir el poder de un lenguaje de programación.

Sin embargo, una restricción podría justificarse por otra, conflicto, consideraciones de diseño etc.

Ejemplos:

- **Se pueden pasar funciones como argumento**

```
map par [1,1,2,3];    map cuadrado [1,1,2,3]
```

- **Se pueden devolver funciones como resultado de otra función**

```
(may 4) 7;            map (may 4) [3,3,6,7,8,3]
```

- **Se pueden almacenar funciones dentro de otras estructuras**

```
mapping [cuadrado, doble] 4
```

Sistema de tipos

Los lenguajes de programación clásicos, tales como Pascal, tienen un sistema de tipos muy simple. Cada constante, variable, resultado de función y parámetro formal, deben declararse con un tipo específico. Este sistema de tipos se denomina **monomórfico**, y hace que el control de tipos sea sencillo.

Pregunta mucho en cuanto a una comparación de los dos sistemas de tipos, monomórfico y polimórfico.

Desafortunadamente, la experiencia muestra que un sistema de tipos monomórfico puro**, es insatisfactorio, especialmente para escribir software reusable. Un sistema de tipos monomórfico es aquel en dónde las constantes, variables, parámetros y resultado de función, tienen un único tipo**. El sistema de tipos de Pascal es básicamente monomórfico.

Muchos algoritmos estándares (tales como los algoritmos de ordenación) son inherentemente estándares, en el sentido de que solo dependen del tipo de valores que se están manejando. Un sistema de tipos monomórfico obliga a declarar los valores que se manejan de un tipo particular.

Este y otros problemas, obligan a desarrollar sistemas de tipos más poderosos, los cuales están implementados en lenguajes modernos como Ada y ML. Los conceptos relevantes son sobrecarga, que es la habilidad de un identificador u operador a denotar varias abstracciones simultáneamente; polimorfismo, que tiene que ver abstracciones que pueden operar uniformemente sobre valores de diferentes tipos y herencia, que se refiere al hecho de que subtipos hereden características de supertipos.

Pascal fuerza a que se especifique el tipo exacto de todos los parámetros formales y del resultado de toda función. Como consecuencia, todas las funciones y procedimientos definidos en Pascal son monomórficas. Sin embargo, ni Pascal ni otros lenguajes son estrictamente monomórficos; por ejemplo, muchas de las funciones o procedimientos predefinidos del Pascal, tienen un tratamiento especial por el compilador.

Polimorfismo es una propiedad de una única abstracción que tiene una (amplia) gama de tipos relacionados; la abstracción opera uniformemente sobre sus argumentos cualquiera sea su tipo.

- Sistema Polimórfico
  - Polimorfismo paramétrico,
  - Polimorfismo de Inclusión,
  - Sobrecarga.
  - Tipos parametrizados,
  - Conversiones de Tipos

En un sistema de tipos polimórfico, podemos escribir abstracciones que operan uniformemente sobre argumentos de una amplia gama de tipos relacionados. ML fue el primer lenguaje de programación que tuvo un sistema de tipos polimórfico.

Abstracciones polimórficas

En ML es simple definir una función polimórfica. La clave es definir el tipo de tales funciones usando tipos variables, en vez de tipos específicos.

Ejemplo 1: La siguiente función acepta un par de enteros y retorna su suma:

```
fun sum (x: int, y: int) = x + y
```

Esta función es del tipo Integer x Integer → Integer. La llamada a la función sum (13, 21) retornará 34.

Ahora considere la función siguiente, la cual acepta un par de enteros y simplemente retorna el segundo de ellos.

```
fun segundo (x: int, y: int) = y
```

Esta función también es del tipo Integer x Integer → Integer. La llamada a la función segundo (13, 21) retornará 21. La llamada segundo (13, true) debería ser ilegal porque el par de argumentos no consiste de dos enteros. La llamada segundo (13) o segundo (1859, 2, 21) también debería ser ilegal, porque el argumento no es un par.

Pero ¿por qué restringir a segundo a aceptar un par de enteros? No hay nada en el cuerpo de la función que es específicamente una operación sobre enteros, por lo tanto el argumento de la función podría, en principio, ser un par de valores cualesquiera. Eso es de hecho posible al definir segundo de la siguiente manera:

```
fun segundo (x: σ, y: τ) = y
```

Esta función se dice que es del tipo σ x τ → τ. Aquí σ y τ se entienden como algún tipo cualquiera.

Ahora la llamada a la función segundo (13, true) es legal. Su tipo se determina como sigue: podemos asociar el tipo de argumentos Integer x valor_verdad al tipo de la función σ x τ → τ. Por lo tanto al sustituir sistemáticamente se entiende que el tipo del resultado de la función será un valor_verdad (true en este caso).

La función segundo es polimórfica. **Esto no implica que aceptará cualquier argumento.** Una llamada como segundo (13) o segundo (1859, 2, 21) todavía son inválidas, porque el argumento no puede asociarse con el tipo $\sigma \times \tau \rightarrow \tau$. Los parámetros permitidos son solo aquellos valores que tienen tipos de la forma $\sigma \times \tau$ (pares).

Ejemplo 2:

Si Pascal permitiese funciones polimórficas, disjunto podría ser escrita de la siguiente manera:

```
function disjunto (s1, s2: set of τ) : boolean;
      begin
            disjunto := (s1 * s2 =[]);
      end;
```

El tipo de la función disjunto es: $\wp\tau \times \wp\tau \rightarrow$ Valor_verdad.

Polimorfismo Paramétrico

Es una propiedad de una única abstracción que tienen una amplia gama de tipos relacionados; la abstracción **opera uniformemente sobre sus argumentos cualquiera sea su tipo**. Algo que se adecue a parámetros de diferentes tipos (polimorfismo de parámetros) Es una única función que sirve para hacer lo mismo para cualquier tipo de parámetro que se le pase.

```
qsort []     = []
qsort (x:xs) =
  qsort smaller ++ [x] ++ qsort larger
  where
    smaller = filter (<=x) xs
    larger  = filter (>x) xs
```

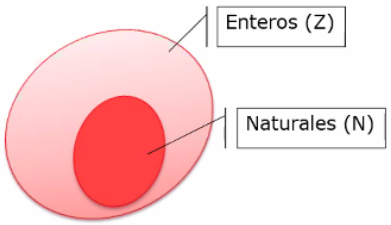
Hay una única abstracción en donde los parámetros no tiene un tipo específico, sino que es genético => el programador puede llamar a esa función con valores de diferentes tipos, en donde la función siempre va a hacer lo mismo.

```
type conjNum = set of 1..10;

function disjunto (s1, s2: conjNum) : boolean;
begin
     disjunto := (s1 * s2 = []);
end;

var
   numeros : conjNum;
begin
   numeros := [9,2,5];
   if disjunto (numeros, [1,2,3])
      then writeln ('Son disjuntos')
   else writeln ('No son disjuntos');
end.
```

El tipo de la función disjunto es: $\wp 1..10 \times \wp 1..10 \rightarrow$ Valor_verdad; y evalúa si el conjunto números es disjunto a [1, 2, 3]. El cuerpo de la función usa operaciones de conjuntos. Sin embargo, la función es monomórfica, no puede aplicarse con argumentos del tipo $\wp$caracteres ni $\wp$color, ni con ningún otro tipo que no sea $\wp$1..10.

Polimorfismo de Inclusión

Habilidad de subtipos y Subclases de heredar operaciones de sus tipos padres o superclases, es decir, es un sistema de tipos donde los tipos pueden tener subtipos que heredan operaciones del tipo padre; específicamente caracteriza a los OOL donde las clases pueden tener subclases que heredan los métodos de la superclase. Una función necesita un numero entero y el programador le pasa uno natural => "error de tipo", pero en realidad si lo acepta.

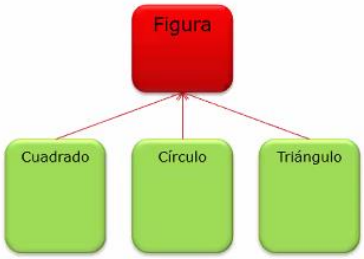


$N \subset Z$; por lo tanto, un valor del subtipo N puede usarse donde se espera un valor del tipo Z.

Z tienen un conjunto de operaciones asociadas que son aplicables también a cualquier valor de N.

Esta técnica se utiliza en los lenguajes orientados a objetos. Se vale de la idea de inclusión.

En general, si C es una clase, esta incluye, no solo objetos de la clase C, sino también objetos de cualquier subclase de C.
Cada objeto de la subclase, potencialmente hereda, todas los métodos de la clase C.



Cualquier puntero a un objeto de Figura puede apuntar a un objeto de cualquiera de sus subclases.

//Clases y objetos → así como los lenguajes de programación que no son OOB definen tipos y valores de un tipo, los OOB definen clases (que se emparejan con los tipos) y definen objetos (que se emparejan con los valores). Este emparejamiento NO es tan directo, tiene cierta relación; así como un tipo establece las características de los valores, las clases hacen +- lo mismo. En los tipos solo ponemos DATOS, en las clases ponemos datos Y programas (son los atributos y métodos). Y los objetos se parecen a los valores, PERO estos tienen los atributos y métodos.//

Se pueden construir clases a través de otras (herencia).

Sobrecarga

Saber definir y diferenciar los tipos de sobrecarga. Relacionarlo con el de objetos.

Sobrecarga significa que un número (pequeño) de abstracciones distintas están **asociadas al mismo identificador**; estas abstracciones no necesariamente deben tener tipos relacionados, ni realizar necesariamente operaciones similares en sus parámetros.

Considere el procedimiento predefinido write, cuya llamada tiene la forma write (E). El efecto de esta llamada a procedimiento depende del tipo de E. Existen varias posibilidades. Por ejemplo, si E es del tipo Char, un simple carácter será escrito. O si E es del tipo string, una secuencia de caracteres deben escribirse, un valor entero deberá convertirse a una cadena de caracteres (un valor decimal rellenado con espacios en blanco) y esa secuencia deberá escribirse. En realidad, el identificador write denota simultáneamente distintos procedimientos. Cada uno con su propio tipo. Este es un ejemplo de sobrecarga.

Pareciese que **la sobrecarga no aumenta el poder de un lenguaje**, debido a que puede ser fácilmente eliminada renombrando las abstracciones sobrecargadas. Así el procedimiento write del Pascal podría renombrarse por writechar, writestring, etc.

La técnica de sobrecarga intenta que varias funciones tengan un mismo nombre => ese identificador se "sobrecarga". Puede haber dos códigos asociados a una función, por ejemplo. A la larga se busca poder valorar diferentes tipos para no tener errores; ayuda en evitar errores de tipos donde no debería haber.

Un identificador u operador **se dice que está sobrecargado** si denota simultáneamente dos o más funciones distintas; cuando tiene asociadas diferentes abstracciones. En general, la sobrecarga es aceptable solo cuando cada llamada a función **no es ambigua**; donde la función a ser llamada puede identificarse unívocamente usando la información de tipos disponible.

Ejemplo 1: el operador '-' denota simultáneamente cinco funciones distintas.

→ Negación de un entero (una función de Integer → Integer)
→ Negación de un real (una función de Real → Real)
→ Diferencia de enteros (una función de Integer x Integer → Integer)
→ Diferencia de reales (una función de Real x Real → Real)
→ Diferencia de conjuntos (una función de Set x Set → Set)

No existe ambigüedad. En las llamada a función tales como '–y' y 'x-y', el número de parámetros actuales y sus tipos determinan unívocamente la función que se llamará.

```
div (a: int; b: int) : int;  (1)
----------
div (a: int; b: int) : float;   (2)

-----------

div (7, 2)

sobrecargas dependientes o independientes del contexto!!!
```

AMBIGUO

Podemos caracterizar la sobrecarga en términos de **los tipos de las funciones sobrecargadas**. En pascal y ML, el tipo del parámetro de la función sobrecargada es siempre distinto. En Ada, el tipo del parámetro o del resultado de la función sobrecargada es distinto.

I denota a una función $f1: S1 \rightarrow T1$, $f2: S2 \rightarrow T2$

Mas generalmente, considere un identificador un operador I que denota a una función f1 del tipo S1 → T1 y otra función f2 del tipo S2 → T2. (Note que se cubren los casos en que las funciones tengan varios argumentos: S1 o S2 podrían ser productos cartesianos). Existen dos tipos de sobrecarga:

Sobrecarga independiente del contexto (como en Pascal, C, C++, Java): (ejemplo de la suma) requiere que S1 y S2 sean distintos. Considere la llamada a la función I (E). Si el parámetro actual E es del tipo S1, entonces I denota a f1 y el resultado es del tipo T1; si E es del tipo S2, entonces I denota f2 y el resultado será del tipo T2. Con la sobrecarga dependiente del contexto la función que se llama es identificada unívocamente por el parámetro actual

```
suma (c1, c2: complejo):complejo;
{......}

suma (p1, p2: polinomio):polinomio;
{------}

suma (m1, m2: matriz):matriz;
{||||||}
```

Sobrecarga dependiente del contexto (como en Ada): (ejemplo de la div) requiere que S1 y S2 sean distintos o que T1 y T2 sean distintos. Si S1 y S2 son distintos, la función a llamar puede identificarse como anteriormente. Si S1 y S2 no son distintos, pero T1 y T2 son distintos, el contexto debe tenerse en cuenta para identificar la función que se llamará. Considere la llamada a la función I(E), donde E es del tipo S1 (≡ S2). Si la llamada a la función ocurre en un contexto donde se espera una expresión del tipo T1, entonces I debe denotar a f1; si la llamada a la función ocurre en un contexto donde se espera una expresión del tipo T2, entonces I debe denotar f2. Con la sobrecarga dependiente del contexto, es posible formular expresiones en las cuales la función a llamar no pueda ser identificada unívocamente pero el lenguaje debe prohibir esas expresiones ambiguas

```
div (a: int; b: int) : int;  (1)
----------
div (a: int; b: int) : float;   (2)

-----------

div (7, 2)
```

Tipos parametrizados

Se parametrizan los tipos para evitar errores. Un tipo parametrizado **es un tipo que tiene otro(s) tipo(s) como parámetros**. Por ejemplo en Pascal el tipo file, set y array.

- Solo algunos lenguajes permiten al programador construir tipos parametrizados nuevos:

data MayBe a = Nothing | Just a

data Either a b = Left a | Right b

data Lista a = LVacia | Par (a, Lista a)

data Tree a = Vacio | Rama (Tree a) a (Tree a)

```
data Tree a = Hoja a | Rama (Tree a) (Tree a)
    deriving Show

hojas :: Tree a -> [a]
hojas (Hoja x) = [x]
hojas (Rama left right) = hojas left ++ hojas right


data TreeB a = Vacio | RamaB (TreeB a) a (TreeB a)
    deriving Show

raiz Vacio = error "Arbol vacio"
raiz (RamaB _ n _) = n

tam Vacio = 0
tam (RamaB ai _ ad) = 1 + tam ai + tam ad

profundidad Vacio = 0
profundidad (RamaB ai _ ad) = 1 + max (profundidad ai) (profu

inOrden Vacio = []
inOrden (RamaB ai n ad) = inOrden ai ++ [n] ++ inOrden ad
```

```
Hugs> Vacio
Vacio
Hugs> RamaB Vacio 1 Vacio
RamaB Vacio 1 Vacio
Hugs> RamaB (RamaB Vacio 2 Vacio) 1 Vacio
RamaB (RamaB Vacio 2 Vacio) 1 Vacio
Hugs> RamaB (RamaB Vacio [True] Vacio) [] Vacio
RamaB (RamaB Vacio [True] Vacio) [] Vacio
Hugs> profundidad Vacio
0
```

Por ejemplo podemos definir file of char o file of integer.

Podemos pensar que file es un tipo parametrizado de la forma file of τ.

En un lenguaje monomórfico, solo algunos tipos predefinidos son parametrizados. El programador no pude defnir nuevos tipos parametrizados.

Ejemplo:

En ML podemos hacer la siguiente definición:

type τ par = τ * τ

En esta definición τ actúa como un parámetro que denota un tipo desconocido. Un ejemplo del uso de par podría ser int par. Al sustituir int por τ, vemos que es equivalente a int * int, y denota un tipo en el cual sus valores es un par de enteros. Otro ejemplo podría ser real par, que denota un tipo en el cual cada valor es un par de números reales.

Conversiones de tipos ~~~~~~

Una conversión de tipos es un mapeo entre valores de un tipo a valores de un tipo diferente. Intenta evitar errores del estilo: se define una función raíz cuadrada definida para números reales => si le paso el 3 debe fallar, pero se hace una conversión automática (o no automática) donde convierte el 3 en un flotante (3.0)

- Ejemplos
  - Mapeos de valores a otros matemáticamente iguales (enteros a reales)
  - Mapeos de valores a otros próximos (mediante truncamiento o redondeo)
  - Mapeos 1 a 1 (caracteres a strings de longitud 1)
  - Mapeos totales (caracteres a ASCII)
  - Mapeos parciales (ASCII a caracteres)

Clasificación:

→Cast: es una conversión explicita. En C, C++ y Java, un cast tiene la forma (T)E, produce un mapeo del valor representado por la expresión E (del tipo S) a su correspondiente valor de tipo T.

→Coersion: es un mapeo implícito entre valores de un tipo a valores de un tipo diferente. Ejemplos típicos de coersiones son mapeos de enteros a números reales y mapeos de caracteres a strings de un único carácter. Una coersion se realiza automáticamente cuando el contexto lo demanda.

Considere un contexto en el cual se espera un opeando del tipo T pero se sustituye por un operando del tipo T' (no equivalente a T). El lenguaje de programación puede permitir una coersión en este contexto proveyendo un mapeo del tipo T al T'.

Esto es ilustrado por la expresión Pascal sqrt (n), *donde la función sqrt espera un argumento del tipo real, pero n es de tipo entero*. Existe un mapeo obvio de Integer a Real:

{..., -2 → -2.0, -1 → -1.0, 0 → 0.0, 1 → 1.0, 2 → 2.0, ...} por lo tanto la expresión sqrt (n) es legal.

Variables y Actualizaciones

En lenguajes imperativos, OOL y concurrentes, una variable es el contenedor de un valor; este valor puede ser inspeccionado y actualizado tanto como se desee

Este tema no suele preguntar, pero con el que sigue se puede relacionarlo.

Variable: espacio de memoria donde guardo un valor, un contendor de un valor, donde el valor puede ser almacenado ahí, actualizado o inspeccionado. Son importantes en los lenguajes, porque de alguna manera dan la posibilidad de que determinados valores sean almacenados temporalmente o no en esas estructuras. => Variables de vida corta (se utilizan dentro de un programa en un programa en particular y están vigente en la ejecución de ese código, tales variables son típicamente actualizadas por la asignación) y de vida larga (están asociadas a archivos, bases de datos, etc.)

En computación, una variable es un objeto que contiene un valor, este valor puede ser inspeccionado y actualizado tanto como se desee. Las variables se usan para modelar objetos del mundo real que poseen estados, tales como la población del mondo, la fecha de hoy, el estado del tiempo actual, o la economía de un país.

Nuestra definición de variables es bastante general. Lo más familiar para un programador son las variables que se crean y se usan en un programa particular, tales variables son típicamente actualizadas por la asignación; y son de vida corta. Sin embargo, archivos y bases de datos son también variables. Estas son generalmente de larga vida, existen independientemente de un programa particular.

Introduciremos la idea de almacenamiento para entender como las variables se actualizan. Los medios reales de almacenamiento (tales como memorias y discos) tienen propiedades que son irrelevantes para nosotros (tales como tamaño de palabra, capacidad, convenciones de direccionamiento). Por lo tanto propondremos un **modelo abstracto de almacenamiento**.

Un almacén es una colección de celdas.

Cada celda tiene un estado actual: reservado o no reservado.

Cada celda reservada tiene un contenido actual, que es un valor almacenable o indefinido.

Ejemplo: considere la variable n en un programa Pascal.

1. La declaración de la variable (var n: Integer) causa que alguna celda no reservada cambie su estado a reservada, y su contenido cambie a indefinido.
2. La asignación n:= 0 cambia a cero el contenido de la celda denotada por n.
3. La expresión n+1 retorna uno más que el contenido de la celda denotada por n. La asignación n:= n+1 suma uno al contenido de esa celda.
4. Al final del bloque que contiene la declaración de n, se revierte el estado de la celda denotada por n a no reservado.

Variables Compuestas

El valor de un tipo compuesto consiste de componentes que pueden ser inspeccionados separadamente. Asimismo, una variable de un tipo compuesto consiste de componentes que son variables también y el contenido de esos componentes puede ser inspeccionado y actualizados separadamente. Un valor de un tipo primitivo ocupa una celda simple, pero una variable de un tipo compuesto puede ocupar varias celdas.

Tiempo de Vida de una Variable

Una propiedad de toda variable es que la misma es creada en algún momento definido y puede ser eliminada en otro momento posterior cuando ya no se la necesita. El intervalo entre la creación y la eliminación es llamado tiempo de vida de una variable. Sólo durante este tiempo la variable ocupa espacio de almacenamiento. Luego de que es eliminada la/s celda/s de almacenamiento puede ser utilizada para otro propósito.

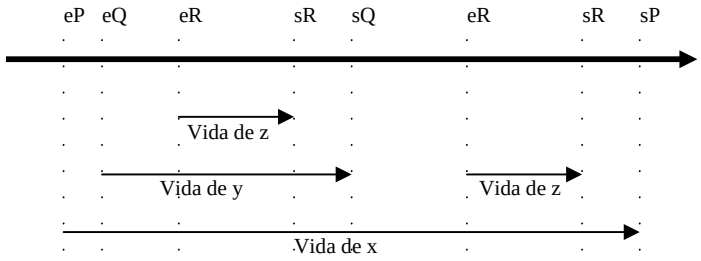
Variables globales y locales

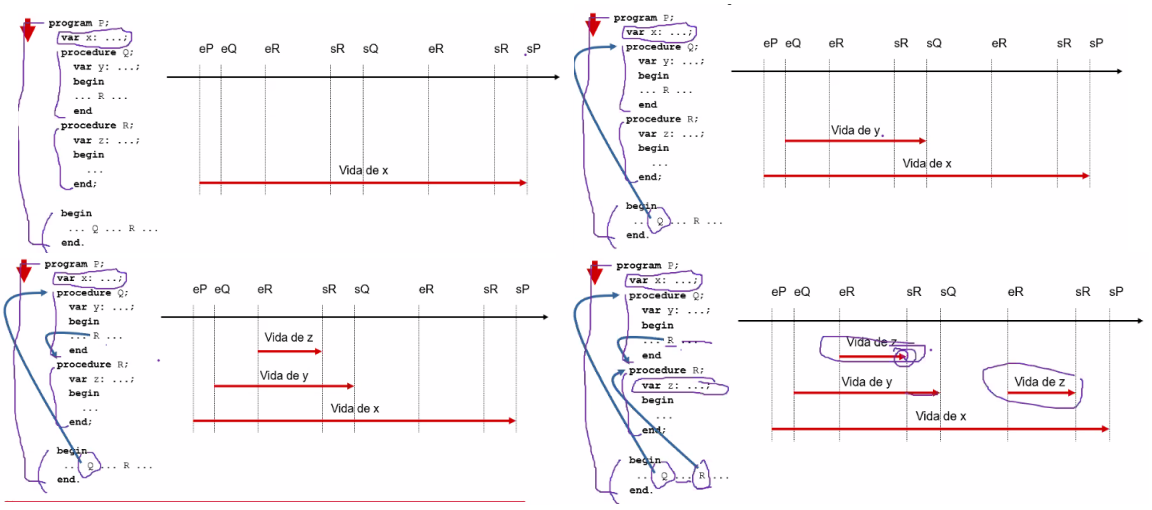
Una variable local es aquella que se declara en un bloque y que se usará solo en dicho bloque (por ejemplo un bloque es el cuerpo de una función o procedimiento en Pascal).

Una variable global es aquella que se declara en el bloque más externo de un programa. La activación de un bloque es el intervalo de tiempo durante el cual ese bloque se ejecuta. En los lenguajes del estilo Algol, el tiempo de vida de una variable local corresponde exactamente a una activación de un bloque que contiene la declaración de la misma.

Considere el siguiente esqueleto de un programa Pascal:

```
program P;
var x: ...;

procedure Q;
var y: ...;
Begin
        ... R ...
end
procedure R;
var z: ...;
begin
   ...
end;

begin
   ...
   Q ...
   R ...
End.
```





Los bloques en el programa son el programa principal P (el bloque más externo), y los procedimientos Q y R. Suponga que P llama a Q y este a R y que a su vez P llama más tarde a R nuevamente. La figura muestra los diferentes tiempos de vida de las variables x, y, z.

Una variable local puede tener diferentes tiempos de vida si el bloque en el que está es activado varias veces. En el ejemplo R es activado en dos momentos distintos por lo tanto los tiempos de vida de z son disjuntos. Si el bloque R es activado recursivamente los tiempos de vida de z serán anidados. Esto tiene sentido si se entiende que cada tiempo de vida de z es realmente el tiempo de vida de una variable distinta, que es creada al inicio de la activación del bloque y eliminada al final de la misma.

Algunos lenguajes como PL/I y C permiten definir variables estáticas (static) cuyo tiempo de vida es toda la ejecución del programa aunque se definan dentro de un bloque interno, aunque solo son alcanzables desde el bloque en que fueron definidas.

Variables de pila (montículo)

Los tiempos de vidas anidados que son característicos de las variables locales y globales es adecuado para muchos propósitos, pero no para todos. A veces es necesario que las variables se creen y se eliminen en función a voluntad.

Este tipo de variables pueden ser creadas y borradas en cualquier momento. Son anónimas y se crean con un comando. Son accedidas a través de un puntero.

Un programa puede construir complejas estructuras de datos, inclusive incorporando a los punteros, que representan las conexiones entre los nodos. Tales estructuras pueden ser actualizadas selectivamente, se pueden agregar y eliminar nodos, cambiar la conexión que existe entre ellos, etc. manipulando los punteros.

Los tiempos de vida de las variables de este tipo no siguen un patrón particular. La creación de una variable de pila se realiza por una operación que retorna un puntero a la variable de pila creada recientemente, la cual estará disponible durante el tiempo en que el puntero lo apunte, hasta que deja de apuntarlo o se la elimine. En Pascal new reserva memoria para una variable de pila y dispose la libera.

Variables Persistentes

Una variable persistente es aquella cuyo tiempo de vida va más allá de la activación de un determinado programa. En contraste una variable transitoria es aquella cuyo tiempo de vida está comprendido en la activación del programa en el cual es creada (var. locales, de pila.)

Los parámetros a un programa son variables persistentes porque existen cuando la activación de un programa comienza y continua hasta después que la activación termina.

Comandos

Habiendo considerado la variables y el almacenamiento, examinemos ahora los comandos. Un comando es una frase de programa que será ejecutada para actualizar variables.

Los comandos son una característica de los lenguajes de programación imperativos (como el Fortran, Cobol, Algol, etc.)

Hay varios tipos de comandos. Unos son comandos primitivos y otros son compuestos de comandos más simples.

Saltos

Es un comando que no tiene efecto alguno (skip). Ej if E then C else skip

Asignaciones

Tienen la típica forma V := E. Actualiza la variable V con el valor retornado por la evaluación de la expresión E.

Ej:
```
(if ... then m else n) := 7
m := n := 0        (asignación multiple)
m, n := n, m       (asignación simultánea)
n += 1             (asignación combinada)
```

Llamadas a procedimientos

Aplica una abstracción de procedimiento a una lista de parámetros.

Comandos secuenciales

El orden en que se ejecutan los comandos es importantes según se actualizan las varibles.

Comandos colaterales

El orden en que se ejecutan los comandos no es irrelevante. Ej: m:= 7, n:= n+1.

Comandos condicionales

Tiene un número de subcomandos de los cuales exactamente uno será elegido para ejecutarse. El típico comando es el if. (case)

Comandos iterativos

También conocido como loop. Tiene un grupo de subcomandos que se ejecutarán repetidamente con un tipo de frase que determina cuando terminará la iteración. (for, repeat, while)

Enlace

Enlace, alcance y ámbito casi no pregunta, pero con un ejemplo pueden definirlos tranquilamente, ya que están relacionados.

Enlace entre nombre (identificador) y entidades. Un identificador no necesariamente está vinculado solo con variables, también con constantes, entre otros.

Un concepto común a todos los lenguajes de programación es la habilidad del programador **para enlazar o relacionar identificadores con entidades** tales como constantes, variables, procedimientos, funciones y tipos. La buena elección de identificadores ayuda a hacer que los programas **sean más comprensibles y fáciles de interpretar.** Más aún, relacionar un identificador con una entidad en un lugar y luego utilizar ese identificador para hacer referencia a esa entidad en otros lugares, **hace que los programas sean más flexibles:** si la implementación de la entidad cambiara, los cambios se deberían hacer en un único lugar y no en todos los lugares donde se usa.

Podemos decir que una declaración produce una asociación o un enlace entre el identificador declarado y la entidad que denotará

Enlace y ámbitos

Considere la expresión n+1 y f(x), y comandos tales como x := 0. No pueden ser interpretados aisladamente. La interpretación depende de lo que denota cada identificador y de cómo fue declarado.

La mayoría de los lenguajes de programación permiten definir un identificador en diferentes partes de un programa, inclusive representando entidades distintas.

Por ejemplo, considere la siguiente declaración Pascal:

```
Const n = 7
```

n representa el número entero 7, por lo tanto la expresión n+1 significa sumar 7 más 1.

Por otro lado, si consideramos la siguiente declaración:

```
Var n: integer;
```

n representa una variable y por lo tanto la expresión n+1 significa sumar uno al contenido actual de la variable n.

Podemos entender los efectos de las declaraciones teniendo en cuenta los conceptos de enlace y ámbitos. Podemos decir que una declaración produce una asociación o un enlace entre el identificador declarado y la entidad que denotará. La definición de la constante del ejemplo anterior relaciona o enlaza 'n' al número 7. La declaración de variable enlaza 'n' a una nueva variable creada.

Un entorno o ámbito es un conjunto de enlaces. Es como el ambiente. Es conocido como entorno o name space.

Cada expresión y comando se interpreta en un ámbito particular y todos los identificadores que se encuentran en la expresión o comando, deben haberse enlazado en ese entorno. Expresiones y comandos en diferentes partes del programa deben interpretarse en diferentes ámbitos. Normalmente, se permite al menos un enlace por identificador en algún ambiente.

Por ejemplo, considere en siguiente programa Pascal:

```
Program p;
      Const z = 0;
      Var   c: char;

      Procedure q;
      Const c = 3.0e6;
      Var b: boolean;

      Begin
            (2)...
      end;

      begin
            (1)...
      end.
```

El ámbito en el punto (1) es:

{ c → una variable de tipo carácter,
q → un procedimiento,
z → el entero 0 }

El ámbito en el punto (2) es:

{ b → una variable que contendrá un valor de verdad,
c → el número real 3000000.0,
q → un procedimiento,
z → un entero 0 }

Una característica de los lenguajes de programación tiene que ver con qué entidades pueden ser enlazados a identificadores. Estas entidades se denominan enlazables del lenguaje. (También pueden llamarse denotables).

Por ejemplo en Pascal son:

→Valores primitivos y strings	en la definición de constantes
→Referencias a variables	en la declaración de variables
→Procedimientos y funciones	en la definición de procedimientos y funciones
→Identificadores de tipos	en las definiciones de tipos.

Note que solo ciertos tipos de valores pueden enlazarse en la definición de constantes en Pascal. Por ejemplo, no se puede enlazar un identificador a un valor de registro, ni a un arreglo (que no sea un string), ni a un conjunto.

Alcance

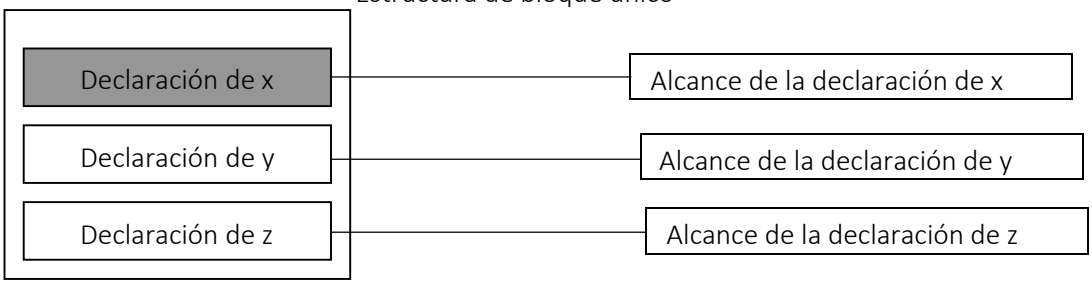
Solo en un lenguaje muy simple cada declaración afecta el ámbito del programa completo. En general cada declaración tiene un cierto alcance, que tiene que ver con la porción del programa sobre el cual al declaración es efectiva. Para entender el concepto introduciremos la idea de estructura de bloque.

Estructura de bloque

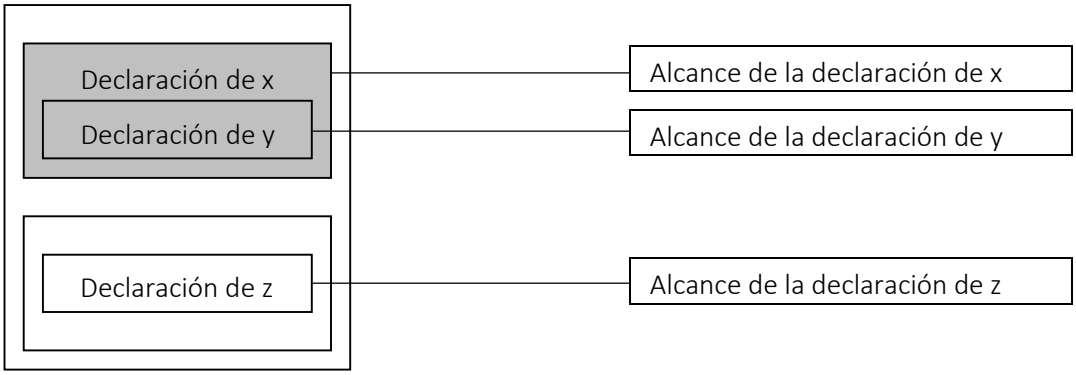
Un bloque es una frase del programa que delimita el alcance de cualquier declaración que puede contener. En particular centraremos nuestra atención en como bloques pueden ser anidados en otros bloques.

Las figuras siguientes muestran los tres tipos de estructura de bloques. Cada bloque está representado por un recuadro. El alcance de algunas declaraciones está sombreadas.

Estructura de bloque único



Estructura de bloque plano



Estructura de bloque anidado

¿Qué ocurre si hacemos una nueva declaración de 'y' en un bloque interior? Ocurre que se hace "invisible por un tiempo" por lo que es nueva declaración tapa temporalmente al bloque de y. El alcance de mantiene pero si hace referencia y en el bloque más pequeño, se refiere a ese pequeño, no al grande.

La estructura de bloque más simple posible es la de bloque único en la cual el programa entero es un bloque simple. Por ejemplo viejas versiones de Cobol. El alcance de cualquier declaración es el programa completo. Todas las declaraciones deben agruparse en el mismo lugar, esto permite asegurarse que todas las entidades declaradas tengan distintos identificadores.

El *segundo caso*, en donde se tiene una estructura de bloques plano, consiste en que el programa es particionado en distintos bloques. El Fortran es un lenguaje que implementa este tipo de estructura, en el cual todos los subprogramas están separados y cada uno actúa como un bloque. Una variable pude declararse en un subprograma particular y es entonces local a ese subprograma.

El alcance de los identificadores de subprogramas es el programa completo, así como el alcance de las variables globales es el programa completo. Una desventaja de esta estructura de bloques es que todos los subprogramas y variables globales deben tener distintos identificadores. Otra desventaja es que cualquier variable que no puede ser local a un subprograma en particular es forzada a ser global aun cuando solo es utilizado por un par de subprogramas.

La estructura de bloques más popular es la anidada donde cada bloque puede ser anidado en otro bloque. Los lenguajes tipo Algol utilizan este concepto. Un bloque puede ser colocado conde sea conveniente y los identificadores se declaran dentro de los mismos.

Los bloques pueden ser abstracciones de procedimiento y función pero también pueden ser bloques de comandos.

Supongamos este ejemplo en donde m y n son variables enteras y sus valores son ordenados de tal manera que m contenga el menor valor. En Pascal escribiríamos esto:

```
If m > n then
Begin
        t:= m;
        m:= n;
        n:= t;
end.
```

Pero la variable auxiliar t debe declararse en algún lugar, quizás en el encabezado del programa o del cuerpo de la función o procedimiento aunque sea utilizado solo en este lugar.

Si Pascal contara con el concepto de bloques de comandos podríamos declarar t localmente:

```
If m > n then var t : integer;
begin
        t:= m;
        m:= n;
        n:= t;
end.
```

Esta localización tiende a hacer a los programas más fáciles de entender y de modificar.

Un concepto aún más potente es el de expresiones de bloque. Es una expresión que es evaluada para producir un enlace con un identificador que se usará solo en el bloque donde se define.

ML maneja este concepto de la forma "let D in E end". Supongamos que a, b y c son los lados de un triángulo. La siguiente expresión de bloque retorna el área ese triángulo:

```
let val s = (a + b + c) * 0.5 in sqrt (s * (s-a) * (s-b) * (s-c)) end
```

Alcance y visibilidad

Los identificadores pueden aparecer en dos contextos diferentes. Llamaremos ocurrencia de enlace al punto en donde un identificador se declara, y llamaremos ocurrencia de aplicación a cada aparición del identificador cuando denota a la entidad a la cual fue enlazado.

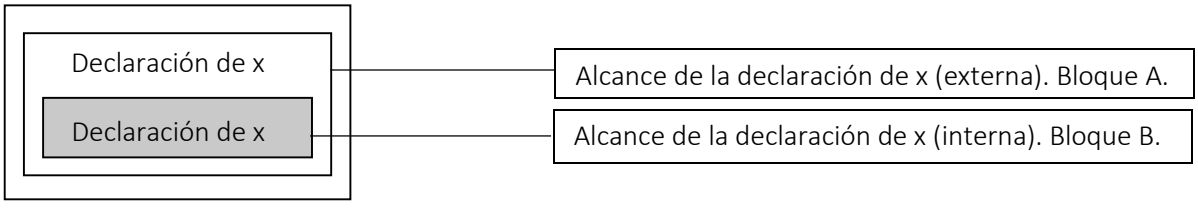
Por ejemplo:

La aparición de n en const n = 7 es una ocurrencia de enlace, en donde n se enlaza al 7.

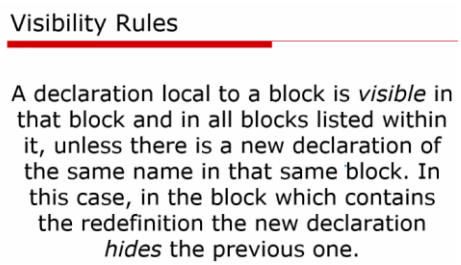
Las apariciones subsecuentes de n en la expresión n * (n+1) son ocurrencias de aplicación, en donde n denota al 7.

Cuando un programa posee más de un bloque, es posible que un identificador se declarado en distintos bloques. En general el identificador puede denotar entidades distintas en cada bloque (aunque no es aconsejable que así lo sea).

Esto da la libertad a los programadores a declarar identificadores sin tener en cuenta si ya fueron declarados en otros bloques. Pero qué pasa si un identificador es declarado en dos bloques anidados (ver figura). La mayoría de los lenguajes con estructura de bloques anidados permiten esto.



Supongamos que dentro del alcance del bloque A está la definición del identificador I, este tiene un alcance en todos los bloques anidados de A y cualquier ocurrencia de aplicación de I se referirá al declarado en A. Si el bloque B posee una declaración del identificador I, cualquier ocurrencia de aplicación dentro del bloque B se referirá al I declarado en B. La declaración de I en el bloque A se dice que es invisible dentro de B o que está oculto por la declaración de I en B. Todas las demás declaraciones en A o dentro del programa principal son visibles dentro de B.



Enlace estático y dinámico

La terminología de enlace estático y dinámico tiene que ver con el momento en que determinamos que ocurrencia de enlace del identificador I se corresponde con la ocurrencia de aplicación de I.

Con enlace estático podemos hacer esto en tiempo de compilación. Podemos determinar que ocurrencia de enlace de I se corresponde con una ocurrencia de aplicación de I dada, solo con examinar el código del programa, encontrando el bloque más pequeño que contiene una ocurrencia de aplicación de I que también contiene una ocurrencia de enlace de I. La asociación entre la ocurrencia de aplicación y enlace es fija.

Con enlace dinámico debemos retardarlo hasta el tiempo de ejecución. La ocurrencia de enlace de I que se corresponde con una ocurrencia de aplicación de I depende del flujo de control dinámico del programa. La entidad denotada por I es la declaración elaborada más recientemente de I dentro del bloque activo actual.

Considere en siguiente código:

```
const s = 2;
function escalar (d: integer): integer;
      begin
            escalar := d * s;
      end;

procedure ... ;
      Const s = 3;
      begin
            (2)... escalar (h);...
      end;

begin
      (1)... escalar (h);...
end.
```

El resultado de escalar (h) depende de cómo interpretemos la ocurrencia de s en el cuerpo de la función escalar.

La interpretación más común es llamada Enlace Estático. Static Scope. Las asociación de entre una ocurrencia de uso de identificado se asocia con la ocurrencia de enlace en tiempo de compilación y no cambian nunca más (pascal, por ejemplo) Se asocia con la declaración más cercana que tiene. El cuerpo de la función es evaluado en el ámbito de la definición de la función. En esta punto s denota 2 por lo tanto la escalar (h) siempre retornará el doble de h, sin tener en cuenta de dónde se llama a la función.

Otra alternativa de interpretación es llamada Enlace Dinámico. Dinamic scope. Las asociaciones se hacen a medida que se ejecuta el código. El cuerpo de la función es evaluado en el ámbito de la llamada a la función. Por lo tanto la llamada a la función escalar (h) en el punto (1) retornará el doble de h debido a que s denota 2 en este punto. Por otro lado la llamada a la función escalar (h) en el punto (2) retornará el triple de h debido a que s denota 3 en este punto.

Es evidente que el **enlace dinámico no encaja con el control d tipos estático.** Si consideramos nuevamente el ejemplo anterior y reemplazamos la definición global de s por const s = 0.5, con el enlace estático se detectará un error de tipos en escalar := d * s. Con el enlace dinámico la llamada a la función en el punto (2) no será afectada pero en el punto (1), producirá un error de tipos en tiempo de ejecución. Por esta razón los lenguajes con enlace dinámico (como el Lisp y Smalltalk) también utilizan el tipeo dinámico. El **problema ilustrado en el enlace dinámico** tiende a hacer que las funciones y procedimientos **sean difíciles de entender**. Si un procedimiento P accede a una constante o variable no local, o llama a una función o procedimiento no local, el efecto dependerá de dónde se llama a P. Debido a esto lenguajes de programación del estilo Algol, optan por el enlace estático.

Abstracciones

Son funciones o bloques de código que realizan determinadas cuestiones o procedimientos/procesos de los cuales uno puede abstraerse.

Cuando hablamos de abstracciones es: cuando le pedimos a alguien que prepare un mate, sin darle indicaciones esa persona lo puede hacer; o pedir que suba o baje una escalera; sin embargo nadie nace sabiendo esas cuestiones. => Son como algo (proceso) que puede ser ejecutado sin enumerar el detalle.

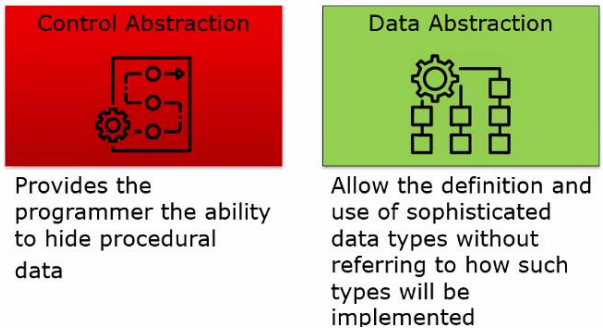
Abstracción es un tema que jode mucho. Tener presente este tema a modo de comparar definiciones o de diferentes paradigmas.

Abstracción es un **modo de pensamiento** en el cual nos concentramos en las ideas generales más que en las manifestaciones específicas de estas ideas. En el análisis de sistemas, la abstracción es la disciplina por la cual nos concentramos en los aspectos esenciales del problema y se ignoran aspectos irrelevantes.

En programación, la abstracción alude a la distinción entre (a) qué hace una parte de un programa y (b) cómo está implementado. Un lenguaje de programación contiene construcciones que son en última instancia abstracciones del lenguaje de máquina. Pero esto no es el final de la historia, se explotan las abstracciones cuando se introduce el concepto de función o procedimiento. Cuando se llama a un procedimiento nos concentramos sólo en qué hace dicho procedimiento; sólo cuando escribimos un procedimiento nos interesará el cómo implementarlo. Al implementar procedimientos de más alto nivel a partir de procedimientos de más bajo nivel, los programadores pueden introducir tantos niveles de abstracción nuevos como deseen.

- Nos concentramos sólo en ***qué*** hace un procedimiento; sólo cuando escribimos un procedimiento nos interesará el ***cómo*** implementarlo.

Calcular o construir funciones, rutinas, sub-programas son abstracciones. => Una abstracción es una entidad que engloba un proceso, una computación.



ABS CONTROL→ son procesos. Nos vamos a enfocar en esta.
ABS DATOS→ Nos pueden servir para modelar algunas cuestiones.

Tipos de abstracción

Definiremos una abstracción como una entidad que engloba una computación. Por ejemplo una abstracción de función engloba una expresión a evaluarse, una abstracción de procedimiento engloba

un comando a ejecutarse. Dicha computación es llevada a cabo cuando la abstracción es llamada. La eficiencia de la abstracción es aumentada con la parametrización.

Las abstracciones tienen parámetro por donde podemos pasarles valores, si no tiene parámetros es una abstracción "muerta", sirve para poco.

```
Vector v = {3.0, 4.0};

float distancia () {
    float s = 0.0;
    for (int i = 0; i < length (v); i++)
        s += v[i]*v[i];
    return sqrt(s);
    }
```

```
float distancia (Vector v) {
    float s = 0.0;
    for (int i = 0; i < length (v); i++)
        s += v[i]*v[i];
    return sqrt(s);
    }
```

Abstracción de función

Una abstracción de función engloba una expresión a evaluarse, y cuando es llamada retorna un valor como resultado. El usuario de la abstracción observa solo su resultado, no los pasos por los cuales se evalúa la función.

Una abstracción de función es construida a partir de una definición de función:

```
Function I (fp1;  ... ; fpn) : T;
      B
```

Donde fpi son los parámetros formales, y donde B es el bloque que contiene al menos un comando de la forma I := E. La abstracción de función se llamará con una expresión del tipo:

I (ap_1, ..., ap_n) donde a_i son los parámetros actuales.

Por ejemplo; considere la siguiente abstracción de función en ML:

```
fun potencia (x: real, n: int) =
      if n = 1 then x
               else x * potencia (x, n - 1)
```

Esta definición enlaza potencia a una abstracción de función. La vista del usuario de esta abstracción de función es que mapea cada par real-entero (x, n) a x^n. La vista del implementador es la manera en cómo se calcula el resultado, como está codificado el algoritmo de cálculo.

Abstracción de procedimiento

Una abstracción de procedimiento engloba un comando a ejecutarse, y cuando es llamado actualizará variables. El usuario de la abstracción de procedimiento observa solo esas actualizaciones y no lo pasos por los cuales fueron efectuados

Una abstracción de procedimiento es construido en Pascal, a partir de una definición de procedimiento:

```
Procedure I (fp1;  ... ; fpn);
      B
```

Donde fp_i son los parámetros formales, y donde B es un bloque. La abstracción de procedimiento se llamará con un comando del tipo:

I (ap_1, ..., ap_n) donde a_i son los parámetros actuales.

Por ejemplo:

```
type ArregloPalabras = array [...] of word;
procedure sort (var words: ArregloPalabras);
      ...
```

Esto enlaza el identificador sort a una abstracción de procedimiento. El cuerpo del procedimiento podría, por ejemplo, implementar el algoritmo de inserción. Esta es la visión del implementador. La visión del usuario es que un comando del tipo sort (diccionario) tendría el efecto de ordenar los valores que contiene el arreglo diccionario. Una implementación del algoritmo QuickSort en la abstracción produciría una ejecución más eficiente, pero el usuario observaría exactamente el mismo efecto neto.

El principio de abstracción

Como un resumen podemos decir:

- **Una abstracción de función** es una abstracción sobre una expresión. Quiere decir que una abstracción de función tiene un cuerpo que es una expresión, y una llamada a la función es una expresión que retornará un valor al evaluar el cuerpo de la abstracción de función.
- **Una abstracción de procedimiento** es una abstracción sobre un comando. Quiere decir que una abstracción de procedimiento tiene un cuerpo que es un comando, y una llamada a procedimiento es un comando que actualizará variables al ejecutar el cuerpo de la abstracción de procedimiento.

Es posible construir abstracciones sobre cualquier clase sintáctica, con tal que las frases de esa clase especifiquen algún tipo de computación.

Por ejemplo, Pascal posee una clase sintáctica para acceso a variables. Un acceso a variable retorna una referencia a una variable. Imagine expandir el lenguaje Pascal diseñando una abstracción sobre acceso a variables. Un tipo de abstracción que cuando es llamada retorna una referencia a una variable. La llamaremos abstracción de selección.

- **Una abstracción de selección** es una abstracción sobre el acceso a variable. Quiere decir que una abstracción de selección tiene un cuerpo que es un acceso a variable, y una llamada al selector es un acceso a variable que retornará una referencia a una variable al evaluar el cuerpo de la abstracción de selección.

Por ejemplo considere las siguientes definiciones Pascal:

```
type cola = ... // cola de enteros

function primero (q: cola) : integer;
        ...   // retorna el primer entero en q
```

Con una llamada a la función del tipo i := primero (UnaCola), permite recuperar el primer entero de UnaCola.

Pero no hay manera de actualizarlo de este modo primero (unaCola) := 0. (Por qué no?)

Suponga ahora que Pascal permite la posibilidad de utilizar abstracciones de selección. Podemos definir Primero para que sea un selector y no una function.

```
selector primero (var q: cola) : integer is
        ...    // retorna una referencia del primer elemento de q
```

Esto permitiría escribir las siguientes llamadas, debido a que el selector retorna una referencia a una variable y no el valor actual.

```
i:= primero (UnaCola);
primero (UnaCola) := primero (UnaCola) + 1
```

Parámetros

Si simplemente hay una expresión dentro de una función o un comando dentro de un procedimiento, construiremos una abstracción que realiza más o menos lo mismo cada vez que es llamada. Para aprovechar la potencia del concepto de abstracción, necesitamos abstracciones parametrizadas respecto a los valores con que opera.

Importante y suele pedir que definas el parámetro y los tipos. Con un ejemplo comparando ambos mecanismos.

Por ejemplo la siguiente abstracción de función en ML:

```
val pi = 3.1416;
```

```
val r = 1.0;
fun circunf () = 2 * pi * r.
```

La llamada a la función circunf() retorna la circunferencia de un circulo de radio 1. En otras palabras, una llamada a esta función siempre realiza la misma computación mientras r esté enlazado al mismo valor.

Podemos hacer que la abstracción de función sea más útil mediante la parametrización con respecto de r.

```
fun circunf (r: real) = 2 * pi * r
```

Ahora podemos llamar a la función con diferentes parámetros y calcular circunferencias de distintos círculos.

Un identificador que se usa en una abstracción para denotar un argumento se **denomina parámetro formal** (a través del cual se puede acceder a un argumento). Una expresión o frase que se pasa como argumento es denominado **parámetro actual** por ejemplo 1.0 o a+b en las llamadas circunf (1.0) y cicunf (a+b) respectivamente. Un **argumento** es el valor u orea entidad que puede pasarse a una abstracción.

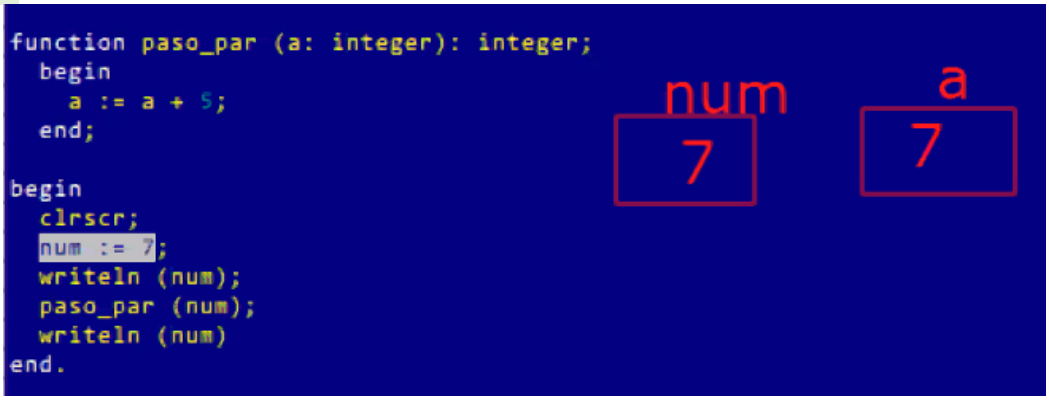
Cuando se llama a abstracción, cada parámetro formal estará asociado, en algún sentido, con su correspondiente argumento.

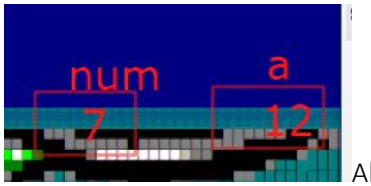
Diferentes lenguajes proveen una variedad de estos mecanismos, por ejemplo parámetros por valor, parámetros resultado, parámetros constantes, parámetros variables etc. Todos estos mecanismos pueden ser entendidos en términos de un pequeño número de conceptos.

Mecanismo de copia

Existe dos mecanismos: un mecanismo de copias (para parámetros) y otro que no hace copias (hace referencias). Por valor el programa cuando lo llamas se entera de lo que le mandas; Por referencia NI se entera. El parámetro formal X denota una variable local para la abstracción. Un valor se copia en X a la entrada de la abstracción, y/o se copia de X (a una variable no local) a la salida de la abstracción. Debido a que X es una variable local, se crea a la entrada de la abstracción y se elimina a la salida.

Un mecanismo de copia permite que valores se copien a y/o desde una abstracción cuando se la llama.





Al escribir num, no sufre cambios gracias a esa copia

Un parámetro por valor es una variable local X que se crea a la entrada de la abstracción y se le asigna el valor del argumento. Debido a que X se comporta como una variable local, su valor puede ser inspeccionado y actualizado. Sin embargo cualquier actualización de X no tiene efecto en ninguna variable no local.

Pascal implementa por valor.

Un parámetro resultado es todo lo contrario del anterior. En este caso, el argumento debe ser una referencia a una variable. De nuevo una variable local X se crea, pero su valor inicial es indefinido. A la salida de la abstracción el valor final de X es asignado a la variable argumento.

Estos dos mecanismos pueden ser combinados para dar resultado a los parámetros valor-resultado. El argumento, de nuevo, debe ser una referencia a variable. En la entrada de la abstracción, la variable local X se crea y se le asigna el valor actual de la variable argumento. A la salida, el valor final de X se asigna nuevamente a la variable argumento.

Por ejemplo, considere el siguiente pseudo código pascal:

```
Type vector = array [1..n] of real;
Procedure sumar (value v, w: vector; result sum: vector);
(1) Var i: 1..n;
      Begin
            For i := 1 to n do  sum[i] := v[i] + w[i];
(2) End;
Procedure normalizar (value result u: vector);
(3) Var i : 1..n; s: real;
      Begin
            s:= 0.0;
            for i := 1 to n do  s := s + sqr (u[i])
            s := sqrt(s)
            for i := 1 to n do u[i] := u[i]/s
(4) end;
```

Supóngase que a, b y c son variables del tipo vector. La llamada al procedimiento sumar (a, b, c) tiene el siguiente efecto: en el punto (1), se crean las variables locales v y w y se les asigna los valores de a y b respectivamente; se crea la variable local sum pero no se inicializa. El cuerpo de la función sumar luego asigna valores a los componentes de sum. En el punto (2), el valor final de sum se asigna a c.

La llamada al procedimiento normalizar(c) tiene el siguiente efecto: en el punto (3), se crea la variable local u y se le asigna el valor de c; el cuerpo de normalizar luego actualiza los componentes de u. En el punto (4), el valor final de u se asigna nuevamente a c.

Debido a que este mecanismo se basa en el concepto de asignación, este mecanismo no es aplicable a tipos donde no se puede aplicar la asignación (por ejemplo archivos en Pascal). Otra desventaja es que la copia de valores compuestos grandes se hace muy cara.

En la siguiente tabla se resume este mecanismo:

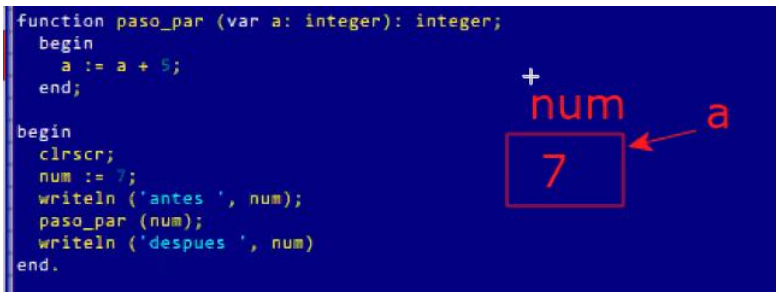
Mecanismo	Argumento	Efecto de entrada	Efecto de salida
Parámetro valor	Valor de 1ª clase	X:= argumento	-
Parámetro resultado	Una variable	-	argumento := X
Parámetro valor-resultado	Una variable	X := argumento	argumento := X

Mecanismo por definición (por referencia)

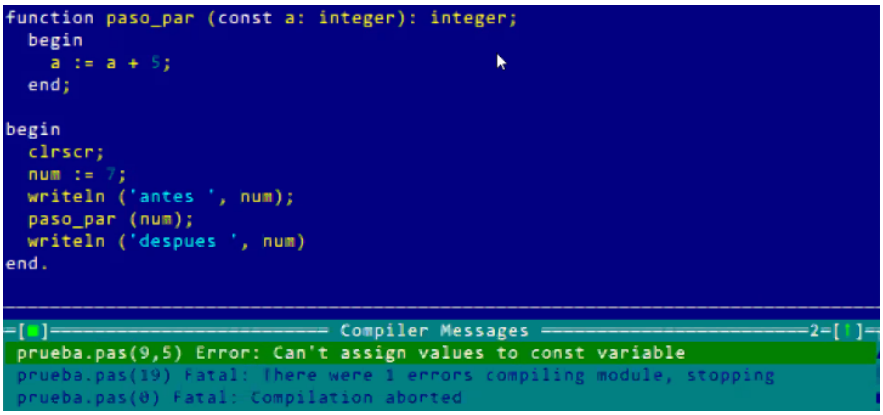
Este mecanismo permite que el parámetro formal X se enlace directamente al argumento. Esto da una semántica simple y uniforme para el paso de parámetros de cualquier valor en el lenguaje de programación (no solo los valores de primera clase). Si el cuerpo de la abstracción esté rodeado por un bloque en el cual hay una definición que enlaza X al argumento — de ahí, nuestra terminología de mecanismo por definición.

- En el caso de un **parámetro constante**, el argumento es un valor (de primera clase). X se enlaza al valor del argumento durante la activación de la llamada a la abstracción.
- En el caso de un **parámetro variable** (o por referencia), el argumento es una referencia a una variable. X se enlaza a la variable argumento durante la activación de la llamada a la abstracción. Por lo tanto cualquier actualización o inspección de X es realmente una actualización o inspección del argumento.
- En el caso de un **parámetro procedural**, el argumento es una abstracción de procedimiento. X se enlaza al procedimiento (argumento) durante la activación de la llamada a la abstracción. Por lo tanto cualquier llamada a X es realmente una llamada al procedimiento (argumento).
- En el caso de un **parámetro funcional**, el argumento es una abstracción de función. X se enlaza a la función (argumento) durante la activación de la llamada a la abstracción. Por lo tanto cualquier llamada a X es realmente una llamada a la función (argumento).

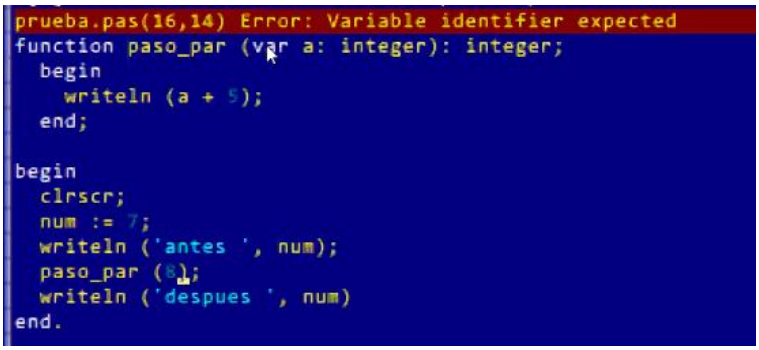
Es importante observar que estos no son mecanismos distintos. En cada caso, el efecto es como si el cuerpo de la abstracción esté rodeado por un bloque en el cual hay una definición que enlaza X al argumento — de ahí, nuestra terminología de mecanismo por definición.



Por referencia, 'a' se vincula a num => si se modifica a, se modifica num. VENTAJA: se ocupa un solo espacio de memoria.



Eso le dice a pascal que, te lo paso por referencia pero NI SE E OCURRA TOCARME el parámetro 'a'. => en lugar de asignarle podría escribirlo, o asignarle a otra variable.



Necesita que le pasemos una variable por referencia, PERO, si en la función cambiamos var por const esta va a compilar sin problemas. Esto es porque resulta que al ser un parámetro consta el compilador se asegura que el valor de a nunca se va a actualizar => le puedo pasar el valor nmas.

Por ejemplo, considere el siguiente pseudo código Pascal similar al ejemplo anterior:

```
Type vector = array [1..n] of real;
Procedure sumar (const v, w: vector; var sum: vector);
(1) var i: 1..n;
      Begin
            For i := 1 to n do  sum[i] := v[i] + w[i];
(2) End;
Procedure normalizar (var u: vector);
(3) Var i : 1..n; s: real;
      Begin
            s:= 0.0;
            for i := 1 to n do  s := s + sqr (u[i])
                  s := sqrt(s)
            for i := 1 to n do u[i] := u[i]/s
(4) end;
```

La llamada al procedimiento sumar (a, b, c) tiene el siguiente efecto: en el punto (1), v y w son enlazados a los valores de a y b respectivamente; y sum es enlazado a la variable c. El cuerpo de suma inspecciona indirectamente los componentes de a y b e indirectamente actualiza los componentes de c.

La llamada al procedimiento normalizar (c) tiene el siguiente efecto: en el punto (3), u se enlaza a la variable c. el cuerpo de normalizar inspecciona y actualiza indirectamente los componentes de c.

Note que no sucede nada en los puntos (2) y (4). Note también que tampoco se necesita una copia para implementar los parámetros constantes. Debido a que los parámetros formales v y w son constantes, no pueden realizar asignaciones a los mismos; de tal manera que los argumentos correspondientes a y b no pueden ser indirectamente actualizados, aún cuando estos parámetros se implementan mediante pasos de referencias de a y b.

Los ejemplos anteriores ilustran el hecho que parámetros constantes y variables proveen un poder similar al mecanismo de copia. La elección de uno de estos mecanismos es una importante decisión de los diseñadores de lenguajes.

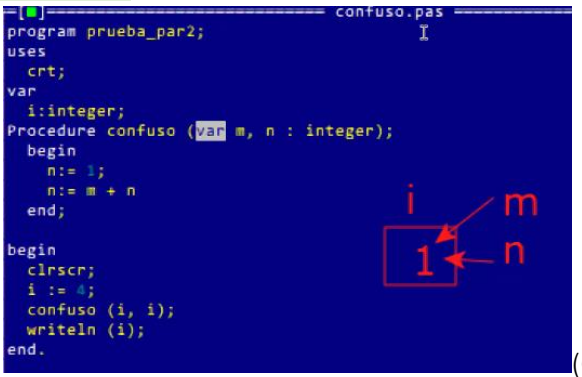
El mecanismo por definición tiene una semántica más simple, y es aplicable a todos los tipos de valores (no solo para los tipos para los cuales la asignación está disponible). Además se apoya en el acceso indirecto de los argumentos, lo cual es a menudo más eficiente que la copia de los datos.

Una desventaja de los parámetros variable es la posibilidad de crear alias. Esto ocurre cuando dos o más identificadores son simultáneamente enlazados a la misma variable. Esto **tiende a hacer que los programas sean más difíciles de entender**. Cuanto más alias escriba, más confusa es la cuestión.

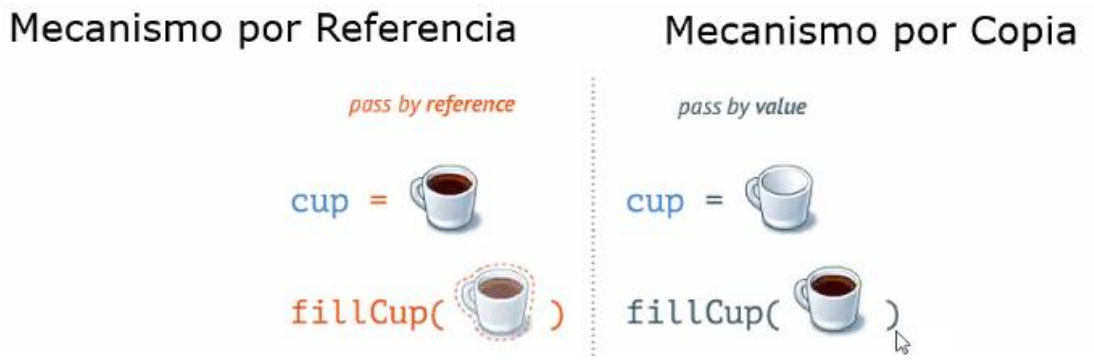
Por ejemplo

```
Procedure confuso (var m, n : integer);
      begin
      n:= 1; n:= m + n
      end;
```

Si la variable i tiene actualmente el valor 4, es de esperar que la llamada a la función confuso (i, i) actualice i a 5 — y sería así si m es un parámetro por valor —. Pero el efecto que se produce es que i se actualiza a 2 !. **Esto es porque m y n son alias de i.** (Porque los dos parámetros hacer referencia al mismo espacio de memoria.)



(Cuando hace n:=1)



Orden de Evaluación

Una función a la cual le pasamos parámetros pero estos no son sencillos, estos parámetros son expresiones que necesitan ser evaluadas para pasar como argumentos. Evaluación impaciente y evaluación perezosa. Los dos métodos siempre dan el mismo resultado

Tema fundamental, suele pedir que compares los tipos de evaluación. Tmb lo compara con el orden de evaluación del cálculo lambda y el por qué usa ese tipo de evaluación (perezosa), a modo de relacionar temas.

```
cuadrado (x : int) -> int
{ x * x }
```

Y la llamada a dicha función cuadrado (p + q). Suponga que el valor de p es 2 y que el valor de q es 5. Hay dos maneras diferentes de evaluar la llamada a la función:

Impaciente (o evaluación en orden aplicativo)→ porque evalúa el parámetro actual de la llamada antes de llamar a la función. Primero evaluar p+q que retornaría 7, y luego enlazar el parámetro formal n a 7. Finalmente evaluar n*n lo cual retornaría 7 x 7 = 49. Se evalúa el parámetro actual una sola vez, y en efecto se sustituye el resultado en cada ocurrencia del parámetro formal.

```
cuadrado (2+3)
cuadrado (5)
5 * 5
25
```

Perezosa (o evaluación en orden normal) → va evaluando, tranqui. Primero enlazar el parámetro formal n a la expresión p+q en sí, luego cada vez que el valor de n es requerido durante la evaluación de n*n, reevaluaremos la expresión a la cual n fue enlazada. Por lo tanto haríamos el siguiente cálculo (2+5)x(2+5) = 49. No se evalúa inmediatamente el parámetro actual, sino que se sustituye el parámetro actual por cada ocurrencia del parámetro formal.

```
cuadrado (2+3)
(2+3) * (2+3)
5 * (2+3)
5 * 5
25
```

El orden de evaluación se refiere exactamente al momento en que cada parámetro actual se evalúa cuando se llama a una abstracción. Existen básicamente dos posibilidades:

- Evaluar el parámetro actual en el punto de la llamada.
- Retardar su evaluación hasta que el argumento realmente se usa.

En el caso de la función cuadrado, ambos tipos de evaluación retornan el mismo resultado **(a pesar de que evaluar en orden impaciente es más eficiente).** Sin embargo algunas funciones pueden producir comportamiento distinto dependiendo del orden de evaluación. Considere el siguiente ejemplo:

```
fun cand (b1, b2 : bool) =
      if b1 then b2 else false
```

Y la llamada a la función cand (n>0, t/n>0.5). Primero suponga que el valor de n es 2 y el de t es 0.8:

- Con evaluación impaciente, n>0 es verdadero, t/n>0.5 es falso, por lo tanto la función retorna falso.
- Con evaluación en orden normal, se evaluará if n>0 then t/n>0.5 else false, lo cual retorna falso.

Pero ahora suponga que el valor de n es 0:

- Con evaluación impaciente, n>0 es falso pero t/n>0.5 falla (por división por 0), por lo tanto la llamada a la función falla.
- Con evaluación en orden normal, se evaluará if n>0 then t/n>0.5 else false, lo cual retorna falso.

La diferencia fundamenteal entre las funciones cuadraro y cand tiene que ver con que si sus argumentos realmente se necesitan. La función cuadrado se dice que es estricta, lo cual significa que una llamada a esta función puede evaluarse solo si sus argumentos pueden ser evaluados. La función cand se dice que es no estricta en su segundo argumento, lo cual significa que una llamada a esta función puede a veces evaluarse aún cuando su segundo argumento pueda no ser evaluado. (La función cand es estricta en su primer argumento)

Función Estricta:
Una llamada a función puede evaluarse solo si sus argumentos pueden ser evaluados.
Función No Estricta:
Una llamada a esta función puede a veces evaluarse aún cuando algún argumento no pueda ser evaluado.

La evaluación en orden normal puede tener algún efecto colateral, por ejemplo:

Supongamos que getint (f) lee un entero de un archivo f y retorna ese entero. Consideremos también la función cuadrado definida arriba y la llamada a la función cuadrado (getint (f)).

- Con evaluación impaciente se leerá un entero y retornaría el cuadrado del mismo.
- Con evaluación en orden normal se leerían dos enteros y se retornaría su producto.

La evaluación en orden normal es claramente ineficiente cuando causa que el mismo argumento sea evaluado varias veces. Si el valor del argumento siempre es el mismo, puede ahorrarse tiempo si el valor del argumento es almacenado tan pronto como sea evaluado por primera vez, y el valor almacenado pueda ser usado cada vez que se necesite. Este evaluación se denomina perezosa (lazy evaluation); el argumento se evalúa solo cuando se la necesita por primera vez (que podría ser nunca si la función es no restricta)

PRESENTACIONES

I - Modularidad

Desde el punto de vista de la programación, podemos decir que la modularidad se trata de la **división en sub-partes de una aplicación o un programa.** Cada una de estas partes está débilmente o fuertemente relacionada con la otra y su funcionamiento es independiente a otra. La modularización incluye las decisiones de diseño que deben tomarse antes que pueda comenzar el trabajo en módulos independientes.

Un módulo es un grupo de componentes declarados para un propósito común. Estos componentes pueden ser tipos, variables, constantes, procedimientos, funciones etc. Tiene un objetivo específico, no tiene que hacer muchas cosas sino que solo debe hacer una. Ya que tiene un propósito y finalidad específico. **Acoplamiento y cohesión**: un módulo cohesivo es que tiene que tener un único propósito y el acoplamiento tiene que ver con la independencia del módulo. Se habla de una mejora cuando se tiene un bajo acoplamiento y una alta cohesión de los módulos. **Ocultamiento**: dentro de un módulo hay cosas que pueden ser privadas, cosas que no pueden ser conocidas en el exterior y solo el modulo sabe que existe; sirven para implementar los componentes públicos.

Un Módulo encapsula sus componentes y permite una interfase con otros módulos y hace conocidos unos pocos componentes hacia fuera del mismo (exportados). Otros componentes quedan ocultos; asisten a la implementación de componentes exportados.

Parnas afirma: *"Las conexiones entre módulos deben seguir el criterio de ocultación de información, es decir, un sistema debe descomponerse bajo el criterio general de que cada módulo oculta alguna decisión de diseño del resto del sistema"*

Por lo que:

- Se reduce la complejidad de cada módulo que compone la solución.
- Al ser partes independientes, reduce la necesidad de tomar decisiones globales.

Como consecuencia de dicha técnica, se pueden generar programas más cortos, legibles y flexibles, permite una organización en el código. Es como una especia de encapsulamiento de alto nivel, que a pesar de ser difícil de lograr es muy importante a la hora de hacer sistemas grandes.

En general, la mayoría de programas que reconocen datos básicos como enteros, reales o booleanos, permiten también la ampliación del lenguaje a través de nuevos tipos. Estos nuevos tipos son los denominados TAD (tipo de dato abstracto). En dicho sentido, la modularización utiliza el concepto de TAD siempre que sea posible.

Si el TAD soporta los tipos que desea el usuario y el conjunto de operaciones sobre cada tipo, se logra un nuevo tipo de dato, denominado objeto.

Desde el punto de vista semántico no hay mucha diferencia entre módulos y TAD, principalmente la habilidad de los módulos de definir más de un tipo a la vez (TAD son un caso particular de módulo). Desde el punto de vista pragmático (implementación) el mecanismo de los módulos permite mayor flexibilidad en cuanto a la permeabilidad (es posible elegir una base cuyas operaciones serán visibles, o elegir el nivel de visibilidad) y además brinda la posibilidad de definir módulos genéricos.

Dado que cada módulo tiene su importancia y significado propio, debe asegurarse que cualquier cambio en su implementación no afecte a su exterior, y asegurar que los posibles errores no se propaguen más allá del módulo o como máximo, los módulos conectados a él.

Para conseguir estas características, se plantean las siguientes reglas:

- Unidades Modulares.
- Pocas Interfaces.
- Interfaces Adecuadas (Pequeñas y explícitas).
- Ocultación de la Información.

Beneficios de la Programación Modular

- El tiempo de desarrollo se acorta porque grupos separados trabajan en cada módulo con poca necesidad de comunicación.

La detección de problemas es más sencillas debido a que el problema se encuentra solamente en una parte del módulo.

Mayor comprensibilidad en la lectura del desarrollo del código y desarrollo del software, ya que debería ser posible estudiar un módulo a la vez de forma individual, esto hace que el sistema esté mejor diseñado.

Una mayor flexibilidad en el producto, ya que realizar cambios drásticos a un módulo no debería de afectar a los demás módulos.

Ventajas de la modularización

Teniendo un punto de vista hacia el programador:

-Divide y vencerás.

Si el desarrollo en el que se trabaja es dividido podemos entonces: Trabajar simultáneamente varias personas, incluso equipos en el mismo proyecto.

-Los programas son más fáciles de documentar y por ende más fáciles de leer y de mantener.

-Mejora la maniobrabilidad del programa

-Permite reutilizar el código.

-Evita colapsos.

Desventajas de la modularización

Cabe recalcar que la modularización requiere de una unión de los módulos y para este tipo de problemas no existe una solución única y funcional.

-La programación modular requiere una inversión de tiempo y dinero extra.

-La unión de los módulos es un reto extra de este método.

-La documentación debe ser precisa para no alterar el correcto funcionamiento del programa en futuras actualizaciones.

-Depurar y probar un código modularizado lleva más tiempo, lo cual conlleva a una menor eficiencia.

2 - Objetos y clases

La Programación Orientada a Objetos es una metodología de diseño de software y un paradigma de programación que define los programas en términos de "clases de objetos", objetos que son entidades que combinan estado (es decir, datos) y comportamiento (esto es, procedimientos o métodos).

Objetos

Un objeto es una entidad que contiene las características o atributos que describen el estado y las acciones que se asocian con un objeto del mundo real, al cual se les asigna un nombre o **identificador.** Es una entidad que tiene un conjunto de responsabilidades y que encapsula un estado interno. Las responsabilidades (o servicios) que realiza el objeto se implementan mediante **métodos**, los cuales describen el comportamiento del objeto. El estado interno del objeto se implementa mediante un conjunto de propiedades o **atributos encapsulados**. Por lo tanto, los objetos combinan procedimientos y datos en una misma entidad.

→ **Estado:** está compuesto de datos, será uno o varios atributos a los que se habrán asignado unos valores concretos (datos).

→ **Comportamiento:** está definido por los procedimientos o métodos con que puede operar dicho objeto, es decir, qué operaciones se pueden realizar con él.

→ **Identidad:** es una propiedad de un objeto que lo diferencia del resto, es su identificador.

Un objeto implementa:

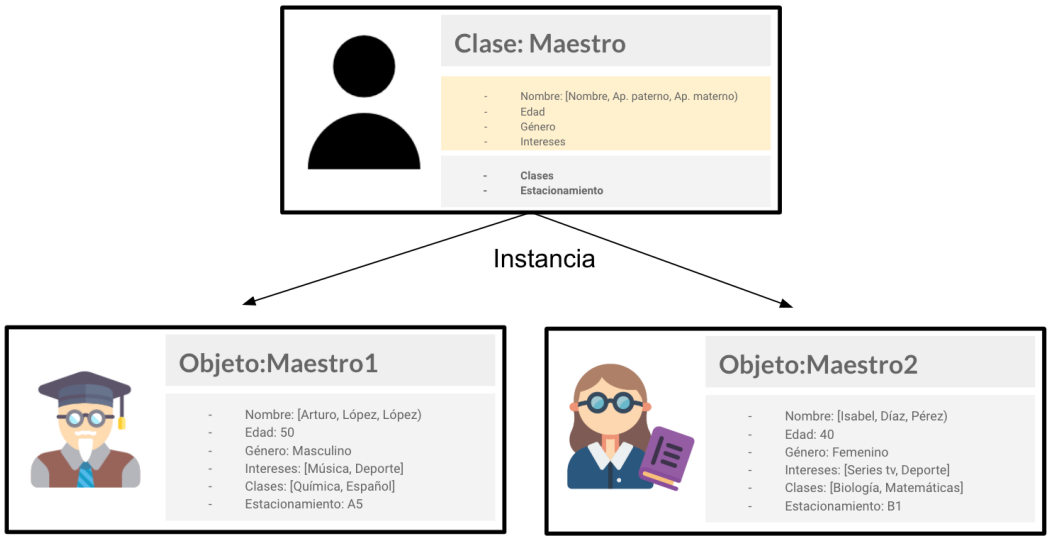
*"**Una abstracción:** denota las características esenciales de un objeto que lo distinguen de todas las otras clases de objetos y que por lo tanto proporcionan limites conceptuales bien definidos, con relación a la perspectiva del observador".*

*"**Encapsulamiento:** es el proceso de esconder todos los detalles de un objeto que no contribuyen a sus características esenciales".*

Clases

Una clase es la descripción de un conjunto de objetos, consta de métodos y datos que resumen características comunes de un conjunto de objetos. Nos permite describir el comportamiento de los objetos y crearlos cuando los necesitemos.

Las clases son similares a los tipos de datos y equivalen a modelos o plantillas que describen cómo se construye ciertos tipos de objetos. Cada vez que se construye un objeto a partir de una clase, estamos creando lo que se llama una **instancia de esa clase**. Por consiguiente, los objetos no son más que instancias de una clase. Una instancia es una variable de tipo objeto. En general, instancia de una clase y objeto son términos intercambiables. Todos los objetos instancias de una misma clase, son estructuralmente iguales; pero pueden tener distintos estados.



Mientras **que un objeto es una entidad concreta** que existe en el tiempo y el espacio, y que lleva a cabo un determinado rol dentro del sistema en general; **una clase representa solo una abstracción**, la "esencia" de un objeto, que captura la estructura y el comportamiento de todos los objetos que a ella pertenecen. Por lo tanto, podemos hablar de la clase Mamífero, que representa las características comunes a todos los mamíferos. Y para identificar a un mamífero en particular en dicha clase, debemos hablar de "este mamífero" o "aquel mamífero".

Para prácticamente todas las aplicaciones, **las clases son estáticas**; por lo tanto, su existencia, semántica y relaciones se fijan en forma previa a la ejecución de un programa. Similarmente, la clase de la mayoría de los objetos es estática, es decir que una vez que un objeto es creado su clase es fija. Por el contrario, **los objetos** son típicamente creados destruidos a una tasa (velocidad) importante durante la ejecución de una aplicación.

3 - Componentes de los objetos y Mensajes. Constructores y Destructores. Miembros de Clase y de Instancia

Componentes de los objetos

Atributos:

Los atributos son las propiedades o estados de un objeto; se declaran como variables y ayudan a crear una estructura predeterminada dentro de ellos. Estos hacen visibles los datos desde fuera del objeto, y su valor puede ser alterado por la ejecución de algún método.

Métodos:

Los métodos son el conjunto de funciones que pueden tener los elementos de un objeto, pueden ser funciones aritméticas, de comparación, de medición, etc. Los métodos pueden ser privados o públicos.

Identidad:

La identidad es la propiedad que permite diferenciar a un objeto y distinguirse de otros. Generalmente esta propiedad es tal, que da nombre al objeto. Tomemos por ejemplo el "verde" como un objeto concreto de una clase color; la propiedad que da identidad única a este objeto es precisamente su "color" verde.

Mensajes

La interacción entre objetos se produce mediante mensajes. Los mensajes son llamados a métodos de un objeto en particular.

Un mensaje está compuesto por los siguientes tres elementos:

1. El objeto destino, hacia el cual el mensaje es enviado.
2. El nombre del método a llamar.
3. Los parámetros solicitados por el método.

En todos los casos los mensajes van dirigidos hacia objetos, NO SE PUEDE TENER MENSAJES HACIA NADA.

Un mensaje consiste en el nombre de una operación y de los argumentos por ella requeridos. Los mensajes constituyen el medio por el cual los objetos se comunican con el fin de lograr una acción cooperativa. Cuando un objeto envía un mensaje a otro objeto, el emisor le está requiriendo al receptor que realice la operación nombrada y (posiblemente) devolverle alguna información. Cuando el receptor recibe el mensaje, realiza la operación requerida en la forma en que sepa hacerla. La petición que hace el emisor no especifica cómo debe ser realizada la operación. Tal información está siempre oculta para el emisor.

Existen tres tipos de mensajes en Smaltalk: unarios, binarios y palabra clave.

Mensajes Unarios → Un mensaje unario es similar a la llamada de una función con parámetro único. Este tipo de mensaje consiste de un nombre de mensaje y un operando. Los objetos se colocan antes del mensaje.

Por lo tanto los mensajes unarios siguen la forma: ObjetoReceptor mensaje

Mensaje Binarios → Los mensajes Binarios son utilizados para especificar operaciones aritméticas, lógicas y de comparación. Un mensaje binario puede ser de uno o dos caracteres de longitud y puede contener cualquier combinación de los siguientes caracteres especiales: + / \ * ~ < > @ % | & ? ! ,

Por lo tanto, los mensajes Binarios siguen la estructura de: ObjetoReceptor caracter ObjetoParametro

Mensaje de Palabra Clave o Keyword → Un mensaje de Palabra Clave es equivalente a una llamada de un procedimiento con dos o más parámetros. Observe el siguiente ejemplo, el nombre del objeto, al cual el mensaje es enviado, se escribe primero, luego el nombre del mensaje (o nombre del método) y luego el parámetro que se pasa. UnObjeto unMensaje:parametro

Los dos puntos (:) son una parte esencial del nombre del mensaje. Cuando hay más de un parámetro, el nombre del mensaje debe aparecer para cada parámetro. Por ejemplo: unObjeto unNombre1:parametro1 unNombre2: parametro2

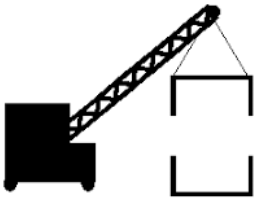
Constructores

Un constructor es un método que se llama automáticamente cada vez que se crea un objeto de una clase. La principal misión del constructor es reservar memoria e inicializar las variables miembros de la clase. Es como una subrutina que nos permite asignar valores iniciales a los objetos. Reservar suficiente espacio en memoria e inicializar los valores de los campos que representan el estado del objeto (atributos y variables).

Características:

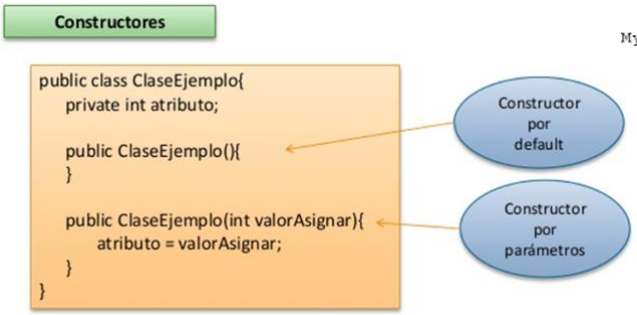
- El constructor siempre lleva el mismo nombre de la clase a la cual pertenece.
- De no definirse un constructor para la clase, el compilador genera un constructor por defecto.
- **El constructor por defecto**, los atributos del objeto se inicializaran con valores predeterminados. (Numéricos con cero, alfanumérico con vacío, y otros objetos con nil)
- **El constructor por parámetros**, el objeto se inicializará con los valores indicados por el usuario.
- Una clase puede tener más de un constructor (sobrecarga).
- No producen retorno.

- Puede existir una relación de herencia.
- Puede especificar cualquier cantidad de parámetros



Constructor

```
MyClass *MyObjPtr = new MyClass();
```

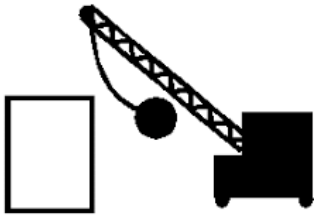


Destructores

Un destructor es un método opuesto a un constructor, éste método se utiliza para destruir una instancia de una clase y libera memoria de nuestra computadora para que pueda ser utilizada por alguna otra variable u objeto. No todos los lenguajes de programación orientados a objetos suelen tener un destructor.

Características:

- Normalmente tienen el mismo nombre de la clase precedido por '~'
- Una clase puede tener un único destructor.
- No pueden heredarse ni sobrecargarse.
- Son invocados automáticamente.
- No acepta modificadores ni parámetros.



Destructor

```
delete MyObjPtr;
```

Miembros de clase

Pueden haber **datos compartidos por todas las instancias de una clase**. Estos se almacenan en variables de clase. Una variable de clase es aquella que solamente tiene un valor para toda la clase y debe ser definida como estática para que no se cree un nuevo valor con cada instancia. Cualquier objeto puede cambiar el valor de una variable de clase y las variables de clase también se pueden manipular sin crear una instancia de la clase.

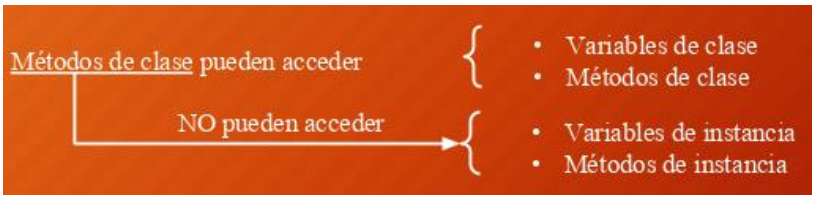
Se puede utilizar inicializadores estáticos para proporcionar valores iniciales para miembros de clase cuando se los declara.

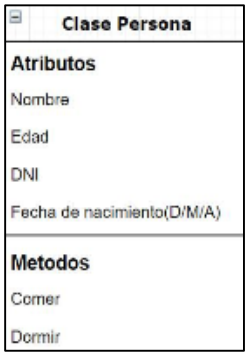
Son de un **comportamiento común** a todas las instancias.

Limitaciones:

- Sólo pueden realizar inicializaciones que puedan expresarse en una declaración de asignación.
- No pueden llamar a ningún método que pueda lanzar una excepción seleccionada.
- Si el inicializador llama a un método que lanza una excepción de tiempo de ejecución, no puede haber error de recuperación.

Proporcionan funcionalidad para la clase.





Un método de clase solo puede actuar sobre variables de clase porque puedo llamar a los métodos de clase aún sin instancias, no puede haber un método de clase que afecte a atributos de clases porque

puede no existir la instancia. Pero un método de instancia puede afectar una variable de clase, porque este ya está creado.

Miembros de instancia.

Pueden haber **datos particulares de cada instancia de una clase.** Estos se almacenan en variables de instancia. Una variable de instancia se relaciona con una única instancia de una clase, normalmente se define como privada, para que no se pueda modificar desde otra clase.

Para inicializar miembros de instancia, se coloca el código de iniciación en un constructor.

Son de un **comportamiento particular** a una única instancia.

Métodos de instancia pueden acceder a:

-Variables de instancia.
-Métodos de instancia.
-Variables de clase.
-Métodos de clase.

4 - Relaciones de Herencia Simple y Múltiple. Conflicto de nombres.

Cuando hablamos de herencia, nos referimos a una de las herramientas más poderosas en POO, la herencia permite modelar de manera precisa la realidad que se desea emular en un programa, abstrayendo el comportamiento y características comunes entre objetos a través de un mecanismo de generalización.

La herencia, junto con la abstracción, encapsulación y el polimorfismo, conforman las 4 características principales (o "pilares") de la programación orientada a objetos. La herencia es específica de este tipo de programación, donde una clase nueva se crea a partir de una clase existente. Proviene del hecho de que la subclase (la nueva clase creada) contiene los atributos y métodos de la clase primaria (clase padre). La principal ventaja de la herencia es la capacidad para definir atributos y métodos nuevos para la subclase, que luego se aplican a los atributos y métodos heredados.

Esta implica la creación de nuevas clases, también llamadas clases derivadas, a partir de clases (clases base). La nueva clase derivada hereda los comportamientos y características de la clase base y también agrega la suya propia. Es una relación entre las implementaciones de dos clases. Se trata de dos mecanismos que son completamente independientes.

La herencia provee un mecanismo para la clasificación, con ayuda de ella se pueden crear taxonomías de clases. La herencia permite concebir una nueva clase de objetos como un refinamiento de otra, con el objeto de abstraerse de las similitudes entre clases, y diseñar y especificar solamente las diferencias para la clase nueva. De esta manera se pueden crear clases rápidamente. También hace posible la reutilización de código. **Este mecanismo de abstracción puede proveer un modo poderoso de producir código que pueda ser reusado una y otra vez**.

Beneficios de la herencia:

- Permite la reusabilidad del código: cuando el comportamiento se hereda de otra clase, no es necesario reescribir el código que lo define.
- Se puede estructurar problemas de la vida real fácilmente
- Facilita la extensibilidad de un programa existente, de manera que realiza nuevas funcionalidades aprovechando las actuales

Redefinición (overriding)

La redefinición se produce cuando una clase vuelve a definir alguno de los métodos o valores heredados de su superclase. El nuevo método sustituye al heredado para todos los objetos de la clase que lo ha redefinido. Sus objetos tienen un comportamiento modificado respecto de los objetos de la superclase.

El método en la subclase redefine y reemplaza al método de la superclase. Hay varias razones por las que puede ser útil redefinir un método: para especificar comportamiento que depende de la subclase, para lograr mayor performance a través de un algoritmo más eficiente para la subclase o guardando un atributo de cálculo dentro de la estructura interna del objeto, etc.

Se pueden redefinir métodos y valores por defecto para los atributos. Toda redefinición debe preservar la cantidad o tipo de parámetros utilizados en un mensaje y el tipo del valor devuelto. **Nunca debe redefinirse un método de manera que sea semánticamente inconsistente con el originalmente heredado**. Una subclase es un caso especial de su superclase y debería ser compatible con ella en todo respecto.

Método de búsqueda

Cuando un objeto de una clase recibe un mensaje, sucede lo siguiente:

-Se busca el método en la clase del objeto receptor

-Si no se lo encuentra, se busca en la superclase inmediata, y así sucesivamente se recorre la jerarquía de clases.

-Si no encuentra el método en ninguna de las superclases, se produce un error porque el objeto no entiende el mensaje

Tipos de herencia

La mayoría de lenguajes de programación modernos, dispone de diferentes tipos de herencia que podemos usar para hacer todavía más eficiente nuestro programa al añadir características o atributos procedentes de diferentes clases.

Herencia Simple → Es el caso más simple, puesto que es aquella en la que una o más clases, subclases o clases derivadas sólo pueden heredar de otra clase (única), es decir que solo tienen 1 clase base o superclase. La herencia simple no puede expresar relaciones múltiples. La jerarquía de herencia es, por tanto, un árbol y decimos que el lenguaje tiene herencia única (o simple).

Herencia Múltiple → Es el mecanismo que permite a una clase heredar de más de una clase, se dice entonces que una clase es una extensión de dos o más clases. Con herencia múltiple se pueden combinar diferentes clases existentes para producir combinaciones de clases que utilizan cada una de sus múltiples superclases. La jerarquía de herencia en tal caso es un grafo orientado a-cíclico. **Ventajas:** permite más libertad en la definición de nuevas clases a partir de las existentes y crear jerarquías de clases completas más fácilmente. **Desventajas:** con frecuencia se produce confusión y ambigüedad.

Jerarquía

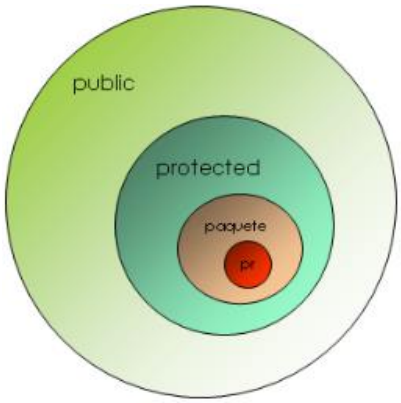
La jerarquía de herencia es también conocida como jerarquía de generalización/especialización. Estos tres términos se refieren a aspectos de la misma idea y frecuentemente son utilizados

indistintamente. Se utiliza generalización para referirse a la relación entre clases, mientras que herencia se refiere al mecanismo de compartir atributos y métodos utilizando la relación de generalización. Generalización y especialización corresponden a dos puntos de vista diferentes de la misma relación, desde la perspectiva de las superclases o de las subclases. La palabra generalización deriva del hecho de que la superclase generalice las subclases. Especialización se refiere al hecho de que las subclases refinan o especializan la superclase.

A medida que la jerarquía de herencia, evoluciona, la estructura interna y el comportamiento que son iguales para clases diferentes tenderá a migrar hacia superclases comunes. Este es el motivo por el cual se dice que la herencia es una jerarquía de generalización/especialización. Las superclases representan abstracciones generalizadas, y las subclases representan especializaciones en las cuales campos y métodos de la superclase son agregados, modificados o aun ocultados. De este modo la herencia permite establecer las abstracciones con una economía de expresión. Hace posible definir software nuevo de la misma forma en que le introducimos conceptos a los novatos, comparándolos con algo que ya les sea familiar.

Herencia y modificadores de acceso

Los modificadores de acceso definen qué clases pueden acceder a un atributo o método. Esto podría servir por ejemplo para ser usados para proteger la información o mejor dicho definir cómo nuestro programa accede a ella. Es decir, los modificadores de acceso afectan a las entidades y los atributos a los que puede acceder dentro de una jerarquía de herencia.



Aunque así de pronto esto pueda parecer complejo lo mejor, para entenderlo, es resumir sus características en una descripción general rápida de los diferentes modificadores

Solo se puede acceder a los atributos o métodos privados (private) dentro de la misma clase.

Se puede acceder a los atributos y métodos sin un modificador de acceso dentro de la misma clase, y por todas las demás clases dentro del mismo paquete.

Se puede acceder a los atributos o métodos protegidos (protected) dentro de la misma clase, por todas las clases dentro del mismo paquete y por todas las subclases.

Todas las clases pueden acceder a los atributos y métodos públicos.

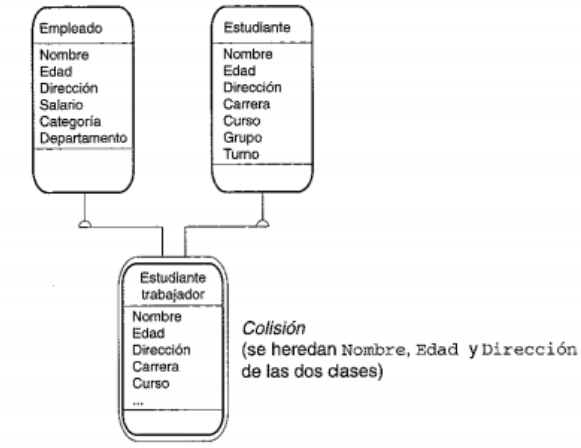
Como puedes ver en esta lista, una subclase puede acceder a todos los atributos y métodos públicos y protegidos de la superclase. Siempre que la subclase y la superclase pertenecen al mismo paquete, la subclase también puede acceder a todos los atributos y métodos privados del paquete de la superclase.

Especificador	Clase	SubClase	Paquete	Todos
public	sí	sí	sí	sí
private	sí	no	no	no
protected	sí	sí	sí	no
no declarado	sí	no	sí	no

Existe una tensión saludable entre los principios de abstracción y encapsulamiento con el de herencia. La abstracción de datos intenta proveer una barrera detrás de la cual métodos y atributos están ocultos; la herencia requiere abrir esa interface hasta cierto punto y puede permitir que tanto métodos como atributos sean accedidos directamente. Para una clase determinada, usualmente hay dos tipos de clientes: los objetos, que invocan operaciones a instancias de la clase, y subclases que heredan de la clase. Con ayuda de la herencia el encapsulamiento puede ser violado de tres formas: la subclase puede acceder a un atributo de la superclase, llamar a un método privado de la superclase, o referirse directamente a la superclase de su superclase.

Colisión de nombres

El conflicto de nombres, se produce específicamente cuando utilizamos la herencia múltiple y ocurre cuando dos o más superclases diferentes tienen el mismo nombre para algún elemento, tales como variables de instancia o métodos.



¿Cómo se puede resolver el conflicto de nombres?

1. Prohibir sintácticamente el conflicto de nombres.
2. Requerir que el programador resuelva los conflictos. Por ejemplo, especificando la clase a la que hace referencia.
3. Elegir una convención para resolver el conflicto. Por ejemplo, dar preferencia a la primera clase nombrada.

Cada lenguaje establece unas reglas al respecto. En el caso de Java:

- Si se repite un atributo: Obliga al programador a especificar de qué interfaz base lo va a utilizar.
- Si se repiten nombres en los métodos: Se dan ciertas condiciones: Si tienen diferentes parámetros se produce sobrecarga de los métodos, así existen diversas alternativas para llamar al método, si solo cambia el valor que devuelve se da un error de compilación indicando que no se pueden implementar simultáneamente y si coinciden en sus declaraciones se elimina uno de los dos.

5 - Relaciones de Agregación y Composición.

Un Sistema con Orientación a Objetos puede ser modelado como un sistema de objetos cooperativos. Algunos objetos solo interactúan con ciertos objetos o tal vez solo con un cierto conjunto de ellos. A veces los objetos son tratados como equivalentes aunque existen diferencias muy específicas entre ellos.

Hay dos tipos de Relaciones: Las relaciones de Agregación y Composición las cuales son tipos especiales de la relación de Asociación. ¿Qué es la Asociación? Es una relación entre dos o más clases que permite asociar objetos que colaboran entre sí para cumplir un objetivo.

Composición ("posee")

Es una **relación por valor**. Modela la noción de un objeto siendo "dueño" de otro, siendo responsable de su creación o destrucción. Los componentes constituyen una parte del objeto compuesto, por eso los componentes **no pueden ser compartidos** por otros objetos compuestos. El objeto base se construye a partir del objeto incluido, por lo que la (al eliminar el objeto se eliminan los componentes) eliminación del objeto compuesto conlleva la supresión de los componentes; es decir, que el tiempo de vida del objeto está condicionado por el tiempo de vida del objeto que lo incluye. Es un tipo de relación estática en la que existe una **relación fuerte**. En UML es un rombo negro.

Agregación ("usa")

Es una **relación por referencia**. Modela la noción de que un objeto usa a otro objeto sin ser su "dueño", entonces no es responsable de su creación ni de su destrucción. La agregación es un tipo de asociación que indica que una clase es parte de otra clase (**relación débil**), los componentes **pueden ser**

compartidos por varios compuestos. La destrucción del compuesto no conlleva la destrucción de los componentes (el tiempo de vida del objeto incluido es independiente del objeto que lo incluye), esto quiere decir que las clases agregadas no afectan el funcionamiento directo de la clase que las contiene. Habitualmente se da con mayor frecuencia que la composición. En UML es un rombo blanco.

La Agregación es un caso especial de asociación. Es una relación parte de en la que los objetos que representan los componentes de algo se asocian con un objeto que representa el agregado completo. La propiedad más significativa de la agregación es la **transitividad**, es decir, si A es parte de B y B es parte de C, entonces A es parte de C. También es **anti-simétrica**, si A es parte de B, B no es parte de A. Algunas propiedades del agregado se propagan a los componentes, posiblemente con modificaciones locales.

	Agregación	Composición
Tipo de relación	Débil	Fuerte
Los componentes comparten varias asociaciones	Si	No
Destrucción de los componentes al destruir el compuesto	No	Si

Ejemplo de Agregación y Composición.

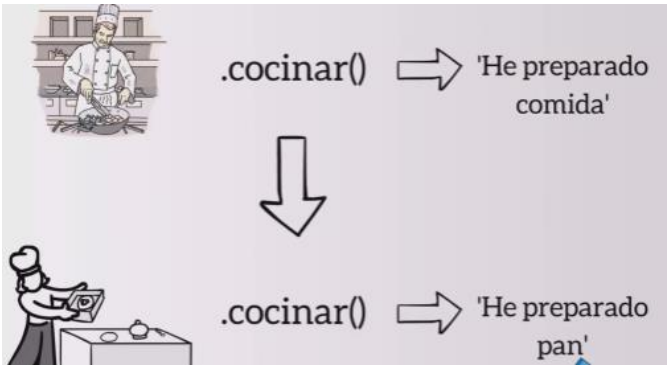
La Universidad está compuesta de facultades, si se destruye la universidad se pierden todas las facultades, es decir, es un ejemplo de Composición. Ahora bien, si se destruye la universidad no afecta la existencia de estudiantes ya que pueden estudiar en otra facultad, es un ejemplo de Agregación.

6 - Redefinición de métodos. Anulación o Sustitución

Un método es una subrutina cuyo código esta definido en una clase. Es la forma de actuar de un objeto.

Redefinición de métodos

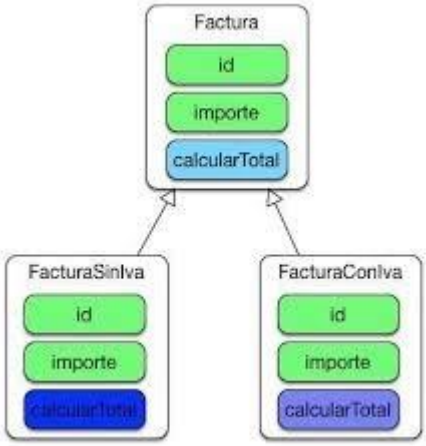
Es una característica que permite que una subclase proporciona una implementación específica de un método heredado de una de sus superclases. Es una característica de Herencia, nos permite cambiar un método heredado. **Ventaja:** ayuda al programador a ahorra código y tiempo ya que superclase ya fue verificada e implementada con anterioridad. También permite que una sub-clase redefina un método definiendo el método con el mismo nombre. **Desventaja:** en el caso de que la jerarquía de clases sea muy compleja esto podría dar complicaciones al entender el programa y el detectar y resolver errores.



Anulación o Sustitución

Definición de Luis joyanes Aguilar*: "Como se ha comentado los atributos y métodos definidos en la superclase se heredan por las subclases, sin embargo si la propiedad se define nuevamente en la subclase, aunque se hayan definido anteriormente a nivel de superclase, entonces la definición realizada en la subclase es la utilizada en esa subclase. Se dice entonces que las propiedades de la superclase se anulan. Esta propiedad se denomina anulación o sustitución."*

En cualquier lenguaje de programación orientado a objetos, Anulación o Overriding es una característica que permite que una subclase o clase secundaria proporcione una implementación específica de un método que ya está provisto por una de sus superclases o clases principales. **Cuando un método en una subclase tiene el mismo nombre, los mismos parámetros y el mismo tipo de devolución** (o subtipo) que un método en su superclase, se dice que el método de la subclase anula el método en la superclase

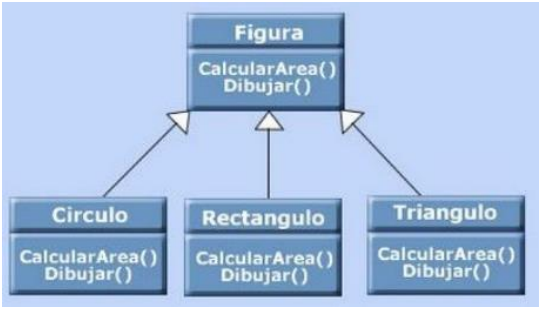


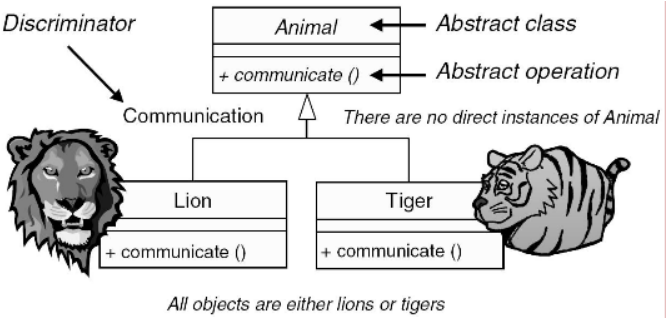
7 - Clases Abstractas y clases concretas

Las clases que no han sido creadas para producir instancias de sí mismas se llaman clases abstractas. Representan un concepto abstracto o entidad con implementación parcial o nula, actúan como clases primarias, de las cuales se derivan las clases secundarias, de modo que la clase secundaria compartirá las características incompletas de la clase primaria. **Sirven para englobar un comportamiento general.**

Existen para que el comportamiento y estructura interna comunes a una determinada variedad de clases pueda ser ubicado en un lugar común, donde pueda ser definido una vez (y más tarde, mejorado o arreglado de una sola vez, sí es necesario), y reusado una y otra vez. Las clases abstractas especifican en forma completa su comportamiento pero no necesitan estar completamente implementadas. Puede también especificar métodos que deban ser redefinidos por cualquier subclase. Un ejemplo de esto puede ser la implementación de cierto comportamiento por defecto para prevenir un error del sistema, pero dicha implementación debe ser redefinida o aumentada por los métodos implementados en las subclases.

Métodos Abstractos → Pertenecen a una clases abstracta. Son métodos sin código, se declaran pero no se definen. Deben definirse en alguna subclase. Si al menos un método de la clase es abstracto, obliga a que toda la clase sea definida como abstracta, no obstante la clase puede tener el resto de sus métodos definidos como no abstractos.



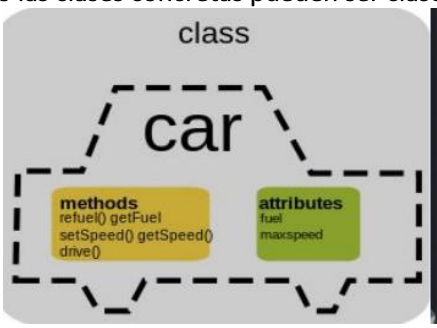


Las **subclases concretas** heredan el comportamiento y estructura interna de sus superclases abstractas, y agregan otras habilidades específicas para sus propósitos. Puede ser que necesiten redefinir la implementación por defecto de sus superclases abstractas, para poder comportarse de alguna forma que tenga sentido para la aplicación de la cual son parte. Estas clases completamente implementadas crean instancias de sí mismas para hacer el trabajo útil en el sistema.

- Son las que pueden usarse para instanciar objetos.

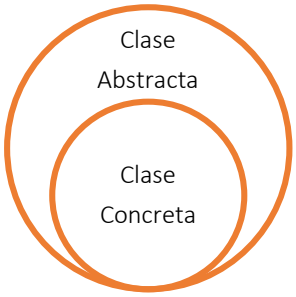
- Tienen todos sus métodos implementados (propios o heredados).
- Son las implementaciones de las Clases Abstractas.

Una clase concreta puede tener subclases abstractas (pero estas a su vez deben tener subclases concretas). **Solo las clases concretas pueden ser clases terminales en el árbol de herencia**.




```
class Coche {

    // Atributos
    public int número_de_ruedas;
    public int litros_gasolina;
    public String modelo;

    // Funciones o métodos
    public void arrancar();
    public void acelerar();
    public void frenar();
}
```

Una clase concreta implementa completamente todos sus métodos. Una clase abstracta se puede considerar como una versión limitada de una clase concreta, donde puede contener métodos implementados parcialmente.

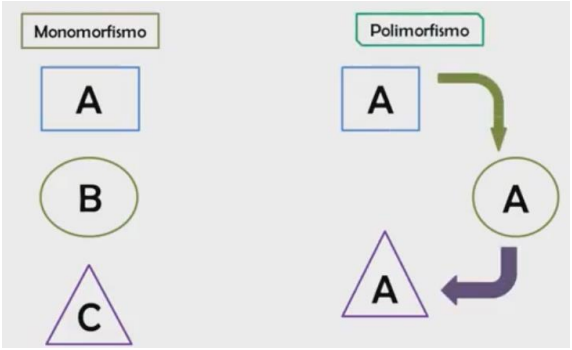


Clase Abstracta	Clase Concreta
Puede extender solo una clase abstracta. Los métodos contenidos deben ser de tipo abstractos para poder implementarla Puede contener constructores. Puede tener modificadores de accesos públicos, privados, estáticos y protegidos. Puede Heredar de una sola Clase Abstracta los Métodos Abstractos. Ideal para reutilizar código No pueden ser Instanciadas.	Una Clase Concreta puede extenderse Se usa para instanciar objetos Sus métodos ya están implementados Puede tener subclases abstractas (pero estas deben tener subclases concretas) Las únicas terminales de un árbol de herencia

TIPO	Clase Base	Clase por defecto
FUNCIONES	Algunas o todas las funciones declaradas son virtuales	No hay funciones puramente virtuales
METODOS	Un método abstracto es un método declarado pero no implementado, es decir, es un método del que solo se escribe su nombre, parámetros y tipo devuelto pero no su código	Todos los métodos están completamente implementados
INSTACIACIÓN	No se puede crear una Instancia porque contiene métodos abstractos, que no tiene un código asociado (solo está el prototipo)	Puede ser instanciado

8 - Polimorfismo

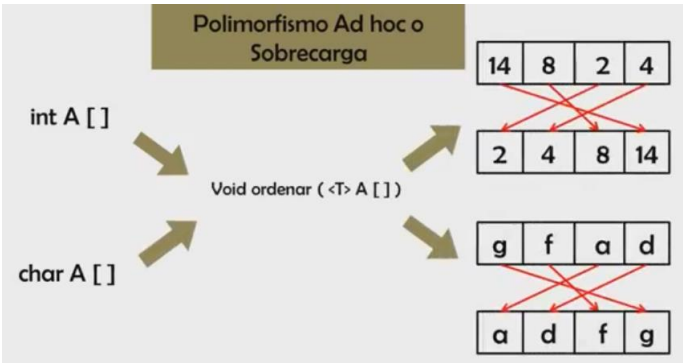
Permite referirse a objetos de clases diferentes mediante el mismo elemento de programa y realizar la misma operación de diferentes formas.



El polimorfismo adquiere su máxima expresión en la derivación o extensión de clases (herencia)

Polimorfismo se aplica a una operación que adopta varias formas de implementación. Es la habilidad de dos o más objetos de responder a un mismo mensaje, cada uno de su propia forma. Esto significa que un objeto no necesita saber a quién le está enviando un mensaje. Solo debe saber que se han definido varios tipos diferentes de objetos para que respondan a ese mensaje en particular.

El Polimorfismo permite reconocer y explotar las similitudes entre diferentes clases de objetos. Cuando reconocemos que varios tipos diferentes de objetos pueden responder al mismo mensaje, estamos reconociendo la distinción entre el nombre del mensaje y un método. Un objeto envía un mensaje: si el receptor implementa un método que corresponde al mensaje, responderá. Diferentes respuestas son posibles, por lo tanto métodos diferentes tienen sentido para clases diferentes, pero el emisor puede simplemente enviar el mensaje sin preocuparse de la clase del receptor. Un concepto relacionado al de Polimorfismo es el de Binding dinámico: significa que el enlace entre el receptor del mensaje, y el mensaje, se realiza en forma dinámica (en tiempo de ejecución).



Cuando hablamos de SOBRECARGA cuando se diferencia el método por el tipo de los parámetros. Cuando hablamos de POLIMORFISMO hablamos de polimorfismo de inclusión, si alguien necesita un numero entero y yo le paso un numero natural me lo debería aceptar sin problema. La herencia proporciona la herramienta necesaria para este tipo de polimorfismo porque representa la inclusión de las sub-clases a la clase padre.

Tipos de Polimorfismo

Polimorfismo paramétrico → Capacidad que permite nombrar a distintos métodos con el mismo nombre. Selecciona de forma automática cual método es más conveniente utilizar.

SUMA
addition(int, int) : int
addition(float, float) : float
addition(str, str) : str

Polimorfismo de Subtipado → La habilidad que le permite redefinir los métodos heredados de una clase base se llama especialización. Las clases derivadas redefinen los métodos y/o propiedades heredados mediante la sobre-escritura (override)