Resumen de la clase 1

Conceptos

- Paradigma
- Programa
- Expresividad
- Abstracción
- Declaratividad
- Imperatividad
- Efecto o efecto colateral

Paradigma

Un **paradigma** es un marco conceptual, un conjunto de ideas.

Un **paradigma de programación** es un estilo fundamental de programación.
Los paradigmas difieren en los conceptos y las abstracciones que utilizan para representar los elementos de un programa y los pasos que componen su evaluación.

Para decir qué es un **programa** podemos tener 2 visiones:

- La **externa**: un usuario cualquiera de computadora puede decir que un programa es algo que logra que la computadora resuelva un problema o necesidad que él tiene. En esta definición de programa pesa el objetivo del mismo.
- La **interna**: como programadores podemos decir que un programa consiste en determinados conceptos (secuencia, condiciones, iteraciones, etc.) y elementos (variables, constantes, memoria, etc.) que se relacionan entre si de una forma determinada (llamadas a funciones, procedimientos, etc.) cuyo objetivo es resolver un problema.

Esta materia se va a tratar de la visión interna de un programa. A medida que vayamos conociendo distintos paradigmas de programación, vamos a ver cómo esa visión interna puede cambiar dependiendo del paradigma que usemos para resolver un problema, y los términos “secuencia, memoria, variable, función” podrían ya no formar parte de la definición de “programa”, o significar algo completamente distinto.

Para cada **paradigma** que trate la materia (Funcional, Lógico y Objetos) se utilizarán distintos **lenguajes** para llevarlos a la práctica (Haskell, Prolog y Smalltalk respectivamente). Será importante aprender los **conceptos** de cada paradigma y los transversales entre ellos independientemente de haber aprendido el lenguaje con el que lo ponemos en práctica, ya que entendiendo bien las ideas de un paradigma se pueden aplicar en cualquier lenguaje que se

base en esas ideas.

Antes de cursar Paradigmas de Programación el alumno debería conocer y haber practicado un paradigma de programación: el **paradigma estructurado** poniéndolo en práctica con lenguajes como C y Pascal.

Declaratividad, abstración y expresividad

Declaratividad

Cuando nuestra solución se basa en determinar una serie de pasos, en donde el concepto de secuencia toma una posición primordial vamos a decir que estamos hablando de una solución "**imperativa**", en donde el programador se concentra en los pasos necesarios para resolver un problema. Aquí entran en juego los conceptos de *algoritmo* y *secuencia*.

Existe otra manera de programar, la "**declarativa**", en donde el programador se concentra en describir las características del problema evitando especificar el orden o el algoritmo[1]. En este tipo de soluciones, el responsable de elegir y ejecutar la secuencia de pasos que resuelve el problema será otro programa llamado **motor**.

Abstracción

Programar implica muchas veces manejar grandes cantidades de información, un programa es una entidad muy compleja y por lo tanto es muy difícil abarcarlo en su totalidad. Por eso necesitamos herramientas que nos permitan manejar esa complejidad.
Gran parte de esas ideas trabajan con el principio de *"dividir y conquistar"*, esto es, tomar una parte más pequeña del programa y poder comprenderla olvidándonos momentáneamente del resto del programa.
Sin embargo, siempre que uno tome una parte de un programa más grande, esa parte tendrá relaciones con otras partes del sistema. Para entenderlo yo debo poder incorporar esas otras partes a mi análisis pero sin tener que comprender todos sus detalles.
Las herramientas que nos permiten eso las denominamos abstracciones.

Una **abstracción** es una forma de interpretar y conceptualizar lo que resulta más importante de una entidad compleja, sin tener que tener en cuenta todos sus detalles. Me permite quedarme con lo *esencial* descartando lo que (para mí, en ese momento) es accesorio.

Una abstracción es un concepto o una idea que no está asociado a ningún caso concreto. Abstraer es formar una idea general a partir de casos particulares. En la vida cotidiana usamos abstracciones todo el tiempo y gracias a eso, por ejemplo, podemos saber que una mesa es una mesa más allá de si es cuadrada o redonda, de madera o de plástico, con 4, 3 o 6 patas.

---

[1] A veces esto no es 100% posible y aparecen algunos componentes algorítmicos, formando una solución *pseudo*-declarativa

Cuando programamos también es importante encontrar buenas abstracciones.

Para los que vienen de programar estructurado, la forma de abstracción más conocida es el procedimiento. Un procedimiento permite tomar un conjunto de instrucciones y darles un nombre para poder utilizarla en otro contexto. Quien invoca el procedimiento se concentra en **qué** es lo que necesita resolver, el procedimiento es el que implementa el **cómo** se resuelve un determinado problema.

Cada uno de los paradigmas que vamos a aprender, nos va a brindar *sus propias formas de abstracción*. Las abstracciones de cada paradigma van a ser conceptos fundamentales en esta materia.

La declaratividad es una forma de abstracción muy poderosa, que nos permite describir el conocimiento relativo a un problema desentendiéndonos de los algoritmos necesarios para manipular esa lógica, que son provistos por el motor.

Expresividad

Un concepto más subjetivo que nos va a importar evaluar en una pieza de código es su **expresividad.** Ésta se relaciona con qué tan fácil nos resulta entenderlo, qué tan descriptivo es el código como para que se “auto-explique”.

¡A no confundirse!

Es importante no confundir los términos de expresividad, abstracción y declaratividad. Aunque están muy relacionados entre sí, **son tres conceptos diferentes**, que bien pueden darse por separado.

La relación que con más frecuencia vamos a encontrar es que un código declarativo tiende a ser más expresivo que uno imperativo, ya que puedo leer directamente de qué se trata el problema, en lugar de deducir qué es lo que un algoritmo está tratando de resolver.

Ejemplo en pseudocódigo:

```
Integer cuantosPares(Array ns){
      Integer i;
      Integer acum = 0;
      for(i = 0 ; i < longitudDelArray(ns) ; i = i + 1){
            if (ns[i] mod 2 == 0) {
                  acum = acum + 1;
            }
      }
return acum;
}
```

Esta función de por sí representa una *abstracción*: es una operación que puedo utilizar cada vez que necesite saber cuántos números hay dentro del array que recibe por parámetro. Gracias a que cuenta con un buen nombre (detalle que aumenta la *expresividad*), podría no tener que leer el código para saber qué es lo que hace.

Leyendo el cuerpo de la función, podemos tratar de encontrar otra abstracción más. En el contexto de la función cuántos pares, ¿qué significa este fragmento?

```
if (ns[i] mod 2 == 0) {
      acum = acum + 1;
}
```

Podemos entender que se está evaluando si ns[i] (el elemento del array que está en la posición i) es múltiplo de 2, y si es así contamos 1 par más. Una *abstracción* podría ser una función que evalúe si un número (en este caso ns[i]) es múltiplo de otro (en este caso 2). El código cambiaría a:

```
if(esMultiploDe(ns[i],2)) {
      acum = acum + 1;
}
```

Al encontrar una *abstracción* y ponerle nombre, hace que mi código quede más *expresivo*. Pero recordemos que hacíamos esto para saber si ns[i] era par. Cuando evaluamos la función esMultiploDe con 2 como segundo argumento, estamos justamente preguntando si ns[i] es par.

Por otro lado, si estamos contando pares, la variable **acum** bien podría llamarse **pares**. Entonces el código podría quedar así:

```
if(esPar(ns[i])) {
      pares = pares + 1;
}
```

Podemos ver que un componente esencial de la expresividad puede ser el **elegir buenos nombres** para los elementos de mi código (variables, funciones, procedimientos, etc).

Haber encontrado estas abstracciones ayudó a que el código quede más expresivo (ya que una "persona normal" entiende más rápido lo que hace `esPar(ns[i])` en comparación con `esMultiploDe(ns[i],2)` o con `ns[i] mod 2 == 0`).
Ahora el código expresa que un número tiene que ser par en vez de exponer el algoritmo o cuentas necesarias para saber si el módulo de la división por 2 es 0.

Esta última solución está más cerca de **qué quiero resolver** en vez de **cómo pretendo resolverlo**.

Efecto o efecto colateral

Entendemos por efecto o efecto colateral a la propiedad que tiene una porción de código de producir cambios que permanezcan más allá de su evaluación, al margen del valor de la expresión en sí.

Buscando ejemplos simples: 2 + 2 es una expresión que no tiene efecto, sólo devuelve un valor. En cambio a := a + 1 tiene un efecto que es incrementar en 1 el valor de la variable a.

Para analizar el efecto de una función o procedimiento debemos ver la función como un todo y pensar en el efecto después de terminar la función, por ejemplo en la siguiente porción de código:

```
Integer siguiente(Integer a){
      Integer b;
      b := a + 1;

      return b;
}
```

Si bien la instrucción `b := a + 1;` tiene el efecto de modificar la variable b, no se considera eso como un efecto de la función `siguiente` dado que la variable b es local a la función y por lo tanto ese cambio no persiste una vez finalizada la ejecución de la función.

En cambio este procedimiento:

```
Integer saldo;
Void pagar(Integer monto){
      saldo := saldo + monto;
}
```

sí tiene un efecto, el de incrementar la variable saldo, que es global y por lo tanto los cambios producidos seguirán vigentes después de finalizado el procedimiento.

En resumen será importante entender cuándo estamos trabajando con código que presente **efecto o efecto colateral**. Una pieza de código tiene efecto si tras su ejecución produjo un cambio visible en el estado del sistema, más allá de su valor de retorno.

Por ejemplo, si dentro de una función cambiamos el valor de una variable global, esa función

tiene efecto, porque el **estado del sistema** ya no es el mismo de antes. Otro ejemplo puede ser una función que imprima algo por pantalla: tras su ejecución la interfaz de usuario cambió y el estado del sistema ya no es el mismo.

Es importante destacar que no todo efecto es negativo, de hecho cuando uno hace un procedimiento como el pagar del ejemplo anterior, el único objetivo de ese procedimiento es producir un efecto. Sin embargo, es muy frecuente que intentemos controlar el efecto colateral en un sistema; para ello solemos dividir a nuestras porciones de código en dos tipos:

- Aquellas que **calculan un valor**, que no suelen tener efecto, porque eso simplifica los cálculos.
- Aquellas que tienen por objetivo **producir un efecto**, que no suelen devolver valores.

Estas ideas no son incompatibles, pueden existir porciones de código (funciones, procedimiento, etc) que devuelvan un valor y además produzcan un efecto; sin embargo en la medida tratamos de evitarlo porque suele ser más complicado trabajar con ellos.

Finalmente, van a notar que este concepto es mencionado de diferentes formas. El nombre original es en inglés *side-effect*, mientras que algunas traducciones al castellano antiguas lo traducían (mal) como *efecto de lado*. En la actualidad nos inclinamos por nombrarlo por *efecto colateral* o simplemente *efecto*.