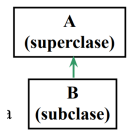
Modulo 1 - Programacion 3

Herencia

El término herencia se refiere al hecho de que una clase hereda los **atributos (variables de estado)** y el **comportamiento (métodos)** de otra clase. La herencia es importante para la **reusabilidad del codigo** en POO.




```
public class Camioneta extends Vehiculos
//Camioneta sera una subclase de Vehiculos
```

*Todas las clases extienden de **Object.***

Subclases

- Pueden agregar comportamiento a la superclase.
- **Sobrescritura de metodos:** Pueden modificar comportamiento heredado.
  - En caso de sobrescritura (si coinciden firmas) se llamara al metodo mas cercano a la subclase. Generalmente es buena practica poner `@Override` .

`Super()` : Invoca a un metodo de la superclase.

```
public String detalles() {
  return super.detalles()+ "\n"
  + "carga máxima:"+getCargaMaxima();
```

Upcasting y Downcasting

Upcasting: Tratar a una referencia de la clase derivada como una referencia de la clase base.

- Se realiza una “conversion hacia arriba”.
- En el ejemplo de debajo vc solo tendra acceso a los metodos propios de Vehiculo. Si quisieramos los metodos de Camion debemos hacerle **Downcasting** a la clase Camion.

```
Vehiculo vc = new Camion();
vc.detalles();
```

Dowcasting: Moverse hacia abajo en la jerarquia del objeto (generalmente haciendole casting).

Clases Abstractas

Una clase abstracta es una clase que **no sera instanciada.** Su principal funcion **definir un comportamiento comun** para las subclases que la **extiendan.**

```
//Clase abstracta
public abstract class FiguraGeometrica
//Metodo abstracto
public abstract void dibujar();
```

Para que las subclases de una clase abstracta **sean concretas**, deben proveer una implementación de cada uno de los método abstractos de la **superclase.**

Listas

Puede estar implementada a través de una estructura estática (arreglo) o una estructura dinámica (usando nodos enlazados).

Tipos Genericos

Permiten al programador **abstraerse de los tipos.** Para poder hacer uso de estos `nombreEstructura<T>`. Luego en momento de Instanciar a estas clases que definen a la estructura definimos el tipo usando los **wrappers.**

Ejemplo: `new nombreEstructura<Integer>()`

Arboles Binarios

Coleccion de nodos compuesta por una **raiz** y dos **posibles hijos.** Si no tiene tal raiz el arbol **es vacio.**

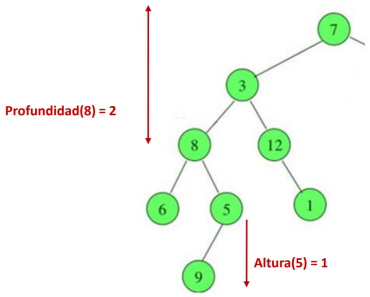
💡 **Hoja:** Nodo que no tiene hijos.

Hermanos: Nodos que comparten padre.

Profundidad(Ni): La longitud del unico camino desde la raiz hasta el nodo Ni.

Grado(Ni): Cantidad de hijos de Ni.

Altura(Ni): Longitud del camino mas largo de Ni hasta una hoja.



Arbol Binario Lleno

Un arbol binario sera **lleno** si cada nodo interno (nodo ≠ hoja) es de grado 2 y las hojas estan todas en el mismo nivel.

Cantidad de Nodos(Arbol):

$$(2^{h+1} - 1)$$

Nodos por Nivel:

Nivel h → 2^h nodos

Arbol Binario Completo

Un arbol binario es completo cuando es lleno hasta la altura (h - 1) y el nivel h se completa de izquierda a derecha.

Cantidad de nodos:

- ***Cantidad de nodos en un árbol binario completo***:

  Sea T un árbol binario completo de altura ***h***, la cantidad de nodos N varía entre (2^{h}) y $(2^{h+1} - 1)$

- Dependera de que cantidad de nodos hojas hay en el arbol.

Recorridos de un Arbol Binario:

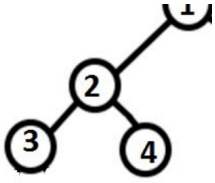
Recorridos en Profundidad:

- **Preorden**
  Se procesa primero la raíz y luego sus hijos, izquierdo y derecho.
- **Inorden**
  Se procesa el hijo izquierdo, luego la raíz y último el hijo derecho
- **Postorden**
  Se procesan primero los hijos, izquierdo y derecho, y luego la raíz
- **Por niveles**
  Se procesan los nodos teniendo en cuenta sus niveles, primero la raíz, luego los hijos, los hijos de éstos, etc.

```
//consiste en 3 pasos donde cambiara unicamente el orden
if(a.tieneHijoIzquierdo()){
  inOrden(a.getHijoIzquierdo());
}
System.out.println(a.getDato()); //Procesado
if(a.tieneHijoDerecho()){
  inOrden(a.getHijoDerecho());
}
//Este ejemplo es de preorden pero cambiando el orden de las ins
```

Recorridos enOrden, preOrden y postOrden.

Recorrido por Niveles:




```
public class ArbolBinario<T> {
    private T dato;
    private ArbolBinario<T> hijoIzquierdo;
    private ArbolBinario<T> hijoDerecho;
    . . .
    public void recorridoPorNiveles() {
        ArbolBinario<T> arbol = null;
        ColaGenerica<ArbolBinario<T>> cola = new ColaGenerica<ArbolBinario<T>>();
        cola.encolar(this);
        cola.encolar(null);
        while (!cola.esVacia()) {
            arbol = cola.desencolar();
            if (arbol != null) {
                System.out.print(arbol.getDato());
                if (arbol.tieneHijoIzquierdo())
                    cola.encolar(arbol.getHijoIzquierdo());
                if (arbol.tieneHijoDerecho())
                    cola.encolar(arbol.getHijoDerecho());
            } else if (!cola.esVacia()) {
                System.out.println();
                cola.encolar(null);
            }
        }
    }
}
```

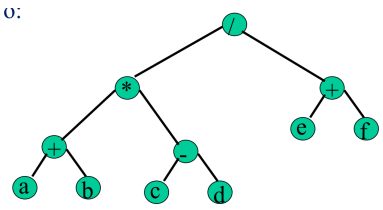
☝ En este ejemplo de Recorrido por niveles se encola una marca de **null** para indicar la finalizacion del nivel. Esta implementacion aplica para arboles generales donde se desea saber cuando termino un nivel.

Arboles de Expresion

Es un tipo de arbol asociado a una operacion aritmetica:

- Los nodos internos son **operadores.**
- Los nodos externos/hojas representan operandos

Los tipos de notacion se dividen en **prefija, sufija e infija.** Cada una se corresponde a un tipo de recorrido.



Construcciones de Arboles de expresion

Expresion Postfija: Mientras sea operando apilo en **Cola. Si**i es un operador desapilo y le agrego eso como HD, vuelvo a desapilar y agrego eso como HI y apilo este nuevo arbol a la **cola.**

Expresion Prefija: Este algoritmo es **recursivo** la diferencia es que aca lo creamos “de arriba hacia abajo”.

```
ArbolExpresión (A: ArbolBin, exp: string )

si exp nulo          → nada.
si es un operador → - creo un nodo raíz R
                      - ArbolExpresión (subArbIzq de R,  exp
                                  (sin 1°carácter) )
                      - ArbolExpresión ( subArbDer de R,  exp
                                  (sin 1°carácter) )
si es un operando → creo un nodo (hoja)
```

Expresion Infija: Convertimos a operacion **Postfija** y luego construimos el arbol de expresion.

(i) Estrategia del Algoritmo para convertir exp. infija en postfija :

a) si es un operando → se coloca en la salida.
b) si es un operador → se maneja una pila según la prioridad del operador en relación al tope de la pila

operador con > prioridad que el tope –> se apila
operador con <= prioridad que el tope –> se desapila elemento colocándolo en la salida.
Se vuelve a comparar el operador con el tope de la pila

c) si es un “(“ , “)”→ “(“ se apila
“)” se desapila todo hasta el “(“, incluído éste

d) cuando se llega al final de la expresión, se desapilan todos los elementos llevándolos a la salida, hasta que la pila quede vacía.

Arboles Generales

Aplican los mismos conceptos que para arboles binarios, con la diferencia en que el **grado** de cada nodo sera variable (antes era 2 como maximo) por lo que la cantidad de nodos en los casos de **arbol completo** y **arbol lleno** tambien cambian.

Cantidad de Nodos:

Arbol Completo: $(k^h + k - 2) / (k-1)$ y $(k^{h+1} - 1) / (k-1)$

Arbol Lleno: $N = (k^{h+1}-1)/(k-1)$

Implementaciones:

- **Lista de Hijos:** Cada nodo tiene informacion propia (dato) y una lista de hijos (cada elemento de la lista es un nodo).

- **Hijo mas izquierdo y hermano derecho**: Cada nodo tiene su informacion propia y un puntero a su hijo mas izquierdo y otro puntero a su hermano derecho.

Recorridos

En profundidad(recursivos)

Pre Orden: Se procesa primero la raiz y luego todos los hijos.

- En este caso el "procesamiento" que realiza el preOrden es el de agregar los datos a una Lista l.

```
private void preOrden(ListaGenerica<T> l) {
  l.agregarFinal(this.getDato());
  ListaGenerica<ArbolGeneral<T>> lHijos = this.getHijos();
  lHijos.comenzar();
  while (!lHijos.fin()) {
       (lHijos.proximo()).preOrden(l);
  }
```

Post orden: Se procesan primero todos los hijos y luego la raiz.

- Para realizar este recorrido alcanza con cambiar el orden del while y `agregarFinal()` del recorrido **preOrden.**

En Orden: Se procesa el primer hijo, luego la raiz y luego los hijos restantes.

Por niveles(Iterativo)

Por Niveles: En el siguiente recorrido se podria haber aplicado la logica de encolar null para indicar **fin de nivel.**

```
public ListaGenerica<T> porNiveles(ArbolGeneral<T> arbol) {

 ListaGenerica<T> result = new ListaEnlazadaGenerica<T>();
 ColaGenerica<ArbolGeneral<T>> cola= new ColaGenerica<ArbolGeneral<T>>();
 ArbolGeneral<T> arbol_aux;
 cola.encolar(arbol);
 while (!cola.esVacia()) {
      arbol_aux = cola.desencolar();
      result.agregarFinal(arbol_aux.getDato());
      if (arbol_aux.tieneHijos()) {
           ListaGenerica<ArbolGeneral<T>> hijos = arbol_aux.getHijos();
           hijos.comenzar();
           while (!hijos.fin()) {
                cola.encolar(hijos.proximo());
           }
      }
 }
 return result;
}
```

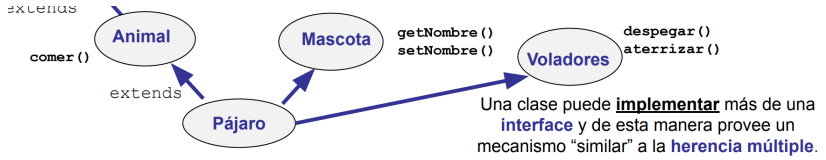
Interfaces

Una interface java es una colección de definiciones de métodos **sin implementación** y de declaraciones de variables de clase constantes, agrupadas
bajo un nombre. La principal ventaja de las interfaces al igual que las clases abstractas es la posibilidad de hacer **upcasting** hacia la interfaz.

Caracteristicas:

- Se provee un mecanismo similar a la "herencia multiple", la cual no es posible en Java.
- Una clase puede implementar mas de una interface.

- Una Interface puede **extender de mas de una interfaz.**
- Las variables seran **implicitamente** `public static final`**.** Mientras que los metodos seran `public abstract`**.**
- Como cualquier metodo **abstract** si los metodos no son implementados el codigo **no compilara.**



Pajaro extiende unicamente de Animal (subclase de Animal) pero implementa las interfaces **Mascota y Voladores**

Declaracion: *(Establecer public para que este disponible para otros packages)*

```
package nomPaquete;
public interface UnaInter extends SuperInter1, SuperInter2, ... {
   Declaración de métodos: implícitamente public y abstract
   Declaración de constantes: implícitamente public, static y final
}
```

- Para denotar que una clase implementa a una interfaz se usa la directiva `implements` luego de la directiva `extends`.

Interfaz vs Clases Abstractas

Diferencias:

- Las interfaces no proveen ninguna implementacion mientras que las clases abstractas podrian implementar algun metodo (al que se le hace referencia con `super()`).
- Con interfaces no hay herencia de métodos, con clases abstractas si.
- Una clase puede extender solo una **clase abstracta** pero puede implementar multiples **interfaces.**

Interface Comparable

Surge ante la necesidad de comparar instancias de objetos en cuanto a **mayor, menor o igual.**

- El metodo implementado por la interfaz comparable es `public int compareTo(T o)`

```
=0: si el objeto receptor es igual al pasado en el argumento.
>0: si el objeto receptor es mayor que el pasado como parámetro.
<0: si el objeto receptor es menor que el pasado como parámetro.
```

Cola de Prioridades

Es una estructura de datos que permite al menos dos operaciones:

- **Insert**
- **DeleteMin**

Implementaciones

??

Heap Binaria

Implementacion de cola de prioridades **sin usar punteros** y permite implementar ambas operaciones en un tiempo $O(log(n))$.

Caracteristicas Estructurales:

- Es un arbol binario completo → altura $O(log(n))$.
- La almacenamos en un arreglo tal que:
  - La raiz es el primer elemento.
  - Para un elemento en la posicion i:
    - Hijo izquierdo esta en `2 * i`
    - Hijo derecho esta en `(2 * i) + 1`. *Al igual que con el hijo izquierdo, si al hacer alguna de estas operaciones nos pasamos es porque no tiene tal hijo.*
    - Padre esta en `i / 2`

Propiedades de Orden (Heap Binaria)

Min Heap:

- Minimo almacenado en la raiz.
- (Dato del nodo ≤ dato de sus hijos).

Max Heap:

- Maximo almacenado en la raiz.
- (Dato del nodo ≥ dato de sus hijos).

Implementacion de Operaciones Basicas

Insertar: Se inserta el dato como ultimo elemento de la heap y luego se hace un filtrado (*percolate up)* para restaurar el orden. Este filtrado tiene un tiempo de ejecucion $O(log(n))$.

DeleteMin: Hago un swap entre la raiz y el ultimo elemento, elimino el ultimo elemento (previamente la raiz) y realizo un filtrado hacia abajo para restaurar orden (percolate down). Este filtrado tiene un tiempo de ejecucion $O(log(n))$.

*Ambas operaciones tienen sus versiones en la **MaxHeap (**en este caso seria un DeleteMax).*

Construccion de Heap (Build Heap)

- Si se insertan los elementos de a uno con la operacion de insertar a la hora de insertar n elementos tendriamos un tiempo de ejecucion $O(nlog(n))$.

La alternativa seria usar un metodo de ejecucion **lineal** (O(n)) llamado **BuildHeap.**

Build Heap:

- Paso 1: Se insertan los elementos en un arreglo desordenado.
- Paso 2: Hacer Percolate Down de sus elementos para establecer orden.
- Se empieza a filtrar desde el elemento con indice `tamanio / 2` ya que el resto son hojas y luego decrementamos indice para “subir” en el arbol.

Ordenacion de Vectores:

- Construir una MinHeap y mientras hacemos deleteMin almacenamos esos mins en el arreglo que quedara ordenado. $O(nlog(n))$
- **HeapSort:** Requiere mismo tiempo de ejecucion pero menos espacio ya que se trabaja sobre el mismo arreglo.
  - Construir una MaxHeap con los elementos que se desean ordenar,intercambiar el último elemento con el primero, decrementar el tamaño (lo hacemos con un indice) de la heap y filtrar hacia abajo.