Estructura de datos son formas de organización de datos.

Cuando un programa se está ejecutando, los objetos necesarios se van creando en la memoria HEAP.

Los objetos que manejan los poo (programas orientados a objetos) se crean a partir de una clase.

Una **clase java** es un bloque de código que describe el estado y el comportamiento que tendrán los objetos que con ella se creen.

Las clases java deben tener en la primera línea la palabra clave Package, seguida del nombre del paquete y la palabra class seguida del nombre de la clase.

Variables de instancia: constituyen el estado de un objeto, normalmente declaradas en private, por lo que solo se puede acceder a ellas desde la clase donde se definen.

Estás están compuestas por

* Un identificador (nombre de la variable)
* Un tipo de dato primitivo o referencial (int, boolean / class)
* Un especificador de acceso (public, private, protected)

Estás están afuera de cualquier método y son creadas cuando el objeto es construido new(). Existen mientras exista el objeto.

Variables locales: se declaran dentro de cualquier método y deben inicializarse antes de ser usadas. Son creadas cuando se comienza a ejecutar y son destruidas cuando el método finaliza su ejecución.

Los parámetros de los métodos también son variables locales y las inicializa el código que llama al método.

Métodos de instancia: definen las operaciones que pueden realizar los objetos de un tipo de clase.

Su declaración especifica

* Un nombre (nombre del método)
* Una lista opcional de argumentos
* Un tipo de retorno
* Un modificador de acceso (public, private, protected)

Tipos primitivos: mantienen valores simples y NO son objetos.

* Entero: byte, short, int, long
* Punto flotante: float y double
* Caracter: char
* Lógico: boolean

Tipos de una clase: referencian a un **objeto** (class) y contienen la ubicacion (dir memoria) de objetos en memoria.

|  |  |  |
| --- | --- | --- |
| 0 | Cliente cli; | Fecha diaCumple = new Fecha(); |
| 1 | cli = new Cliente; |  |

Si la declaración no inicializa las variables de instancia, las mismas toman valores por defecto. Si estas son referencias a objetos, se inicializan con null.

Los datos primitivos no son considerados objetos (por java), se tratan en su forma primitiva por razones de eficiencia.

Java proporciona clases wrappers para manipular los datos primitivos como objetos.

Los **wrappers** son inmutables (también Strings) ya que, para cambiar el valor de estas, se debe crear un nuevo objeto con el nuevo valor.

Para **instanciar una clase** se usa el operador New. Esta retorna una referencia al objeto.

La creación e inicialización de un objeto, es decir, para crear un objeto de una clase

* Se aloca espacio para la variable
* Se aloca espacio para el objeto en la HEAP y se inicializan los atributos con valores por defecto
* Se inicializan explícitamente los atributos del objeto
* Se ejecuta el constructor
* Se asigna la referencia del nuevo objeto a la variable.

Los **constructores** son piezas de código que permiten definir un estado inicial específico de un objeto, en el momento de su creación.

* Tienen el mismo nombre que la clase (pueden empezar con mayúscula)
* No retornan valor
* Se invocan automáticamente

Su inicialización está garantizada, cuando se crea el objeto, se aloca almacenamiento en la memoria HEAP y se invoca al constructor.

Constructor default es aquel que tiene cuerpo vacío. Si el objeto no lo posee, el compilador lo creara automáticamente.

La **sobrecarga de constructores** permite que el mismo nombre de método sea usado con distintos tipos y cantidad de argumentos.

La palabra **static** declara variables y métodos asociados con la clase (no dependen de que la clase sea instanciada, creada).

Las variables de clase son compartidas por todas las instancias de la clase.

Los métodos de clase no dependen de una instancia en particular.

Un **arreglo** es un objeto que hace referencia a un conjunto de valores primitivos u objetos.

Permiten manipular un conjunto de valores del mismo tipo datos usando un único nombre.

Los datos almacenados en un arreglo, se guardan en posiciones contiguas.

Los arreglos tienen una cantidad fija de elementos, una vez creados su dimensión no puede cambiar.

La declaración crea la variable arreglo, no el objeto arreglo.

int[] intArray;

Para crear el arreglo se usa el operador new.

Cuando se crea el objeto se debe indicar la cantidad de elementos o longitud del arreglo.

intArray = new int[5];

Cliente[] cliArray = {new Cliente(), new Cliente()}

Se puede recorrer de 2 formas

For(int i = 0; i < a.length; i++){

Result = result + a[i];

For(int elem: a){ //for-each

Result = result + elto;

La línea for(int elto:a) se lee “para cada elemento ‘elem’ de tipo int, en el arreglo a”

En Java los parámetros se **pasan por valor**, significa que cuando se invoca a un método, se pasan como argumentos al método una copia de cada parámetro actual.

Parámetros formales: son los parámetros en la definición del método.

Public static int mult(int x, int y)

Parámetros reales: son los parámetros en la invocación al método.

Int área = mult(alto, ancho);v

Dentro del cuerpo se puede cambiar el valor de la copia que se recibe como parámetro, pero no tendrá efecto en el parámetro real.

Cuando java llama a un método hace una copia de sus parámetros reales y las envía al método. Cuando finaliza, se descartan.

En java podemos **pasar una referencia** a un objeto y con ella cambiar el valor de sus variables, pero no la referencia.

La poo (programación orientada a objetos) permite a las clases hacer uso de la **herencia**.

La herencia se refiere a que una clase hereda las variables y métodos de otra clase.

La clase que hereda se llama subclase y la clase que es heredada se llama superclase.

A (subclase) B (superclase)

La herencia permite la reusabilidad del código, se hace mediante la palabra clave extends

Public class Camioneta **extends** Vehiculo

Una subclase puede

* Agregar variables y métodos independientes de la superclase
* Reemplaza o modificar el comportamiento heredado (sobreescritura)

Un **método sobrescribe** a otro cuando se define en una subclase y coincide el nombre, tipo de retorno y lista de argumentos con un método ya definido en una superclase.

Es posible invocar el método sin sobrescribir (superclase) desde la subclase mediante la palabra super.

**Upcasting** es convertir una referencia de una subclase a una superclase.

Vehiculo camion = new Camion();

Si la subclase tiene un método que hace override (sobreescribe) a otro, cuando se lo llame, por más que haya hecho Upcasting, va a ejecutar el método de la subclase.

**Downcasting** es convertir una superclase a una subclase.

Animal animal = new Perro(); //upcasting

Perro perro = (Perro) animal; //conversión a tipo Perro (downcasting)

Solo se puede hacer cuando el objeto original es en realidad una instancia de la clase a la que se quiere convertir (en general no se recomienda hacer downcasting).

La **clase Object** es la raíz de todas las clases java, cuando se declara una clase, java implícitamente agregar extends Object a la declaración.

Esta clase posee los métodos equals y toString (se pueden sobrescribir).

**Equals** retorna un valor boolean, compara si 2 referencias son iguales (apuntan al mismo objeto en memoria).

**ToString** retorna la dirección del objeto como un String.

Las **clases abstractas** son clases cuya finalidad es lograr una interface de comportamiento común (molde) para sus subclases, debido a esto su finalidad no es crear instancias (no es instanciable).

Se espera que una clase abstracta sea extendida por clases que implementen todos sus métodos abstractos.

Pueden tener métodos concretos o abstractos.

Los **métodos abstractos** no tienen implementación (están vacíos), se crean con la palabra clave **abstract**. Sirven para definir un comportamiento común a todas las subclases.

**Lista**: Es una estructura de datos en donde los objetos están ubicados en forma secuencial. A diferencia de la **Pila** y la **Cola**, en una Lista se puede agregar y eliminar en cualquier posición.

Puede estar implementada a través de

* Estructura estática (arreglo)
* Estructura dinámica (usando nodos entrelazados)

**Listas de un tipo específico** (private Integer[] datos = new Integer[200])

* Ventajas: El compilador chequea el tipo de dato que se inserta, no es necesario castear.
* Desventajas: Hay que definir una estructura de datos para cada tipo (opcional).

**Listas usando Object** (private Object[] datos = new Object[200])

* Ventajas: Se logra una estructura genérica (deja agregar cualquier tipo de dato).
* Desventajas: El compilador pierde la oportunidad de realizar chequeos, se debe al uso de casting (podría dar error en ejecución).

Un tipo genérico es una variable que le permite al programador abstraerse de los tipos.

Esto permite definir estructuras en las que la especificación del tipo se posterga hasta el momento de la instanciación. Se usa <tipo>

public class ListaEnlazadaGenerica<T> extends ListaGenerica<T>

Private NodoGenerico<T> inicio;

Cuando se instancian las estructuras de datos, se debe definir el tipo de objetos que se vayan a almacenar.

ListaGenerica<Integer> lista = new ListaGenerica<Integer>();

**Árbol binario**: Es una colección de nodos.

* puede estar vacío
* formado por un nodo R, llamado raíz y dos sub-árboles T1 y T2, donde cada subárbol Ti (T1 y T2) está conectado a R por medio de una arista.
* Cada nodo puede tener 2 nodos hijos.
* Cuando un nodo no tiene ningún nodo se denomina hoja.
* Los nodos que tienen el mismo nodo padre se denominan hermanos.

**Camino**: desde n1 hasta nk, es una secuencia de nodos n1,n2,…nk tal que ni es el padre de n(i+1), para i ≤ i < k.

* La longitud del camino es el número de aristas, es decir k-1.
* Existe un camino de longitud cero desde el nodo a sí mismo.
* Existe un único camino desde la raíz a cada nodo.

**Profundidad de ni**: Es la longitud del único camino desde la raíz hasta ni.

* La raíz tiene profundidad cero.

**Grado de ni**: Es el número de hijos del nodo ni.

**Altura de ni**: Es la longitud del camino más largo desde ni hasta una hoja.

* Las hojas tienen altura cero.
* La altura de un árbol es la altura del nodo raíz.

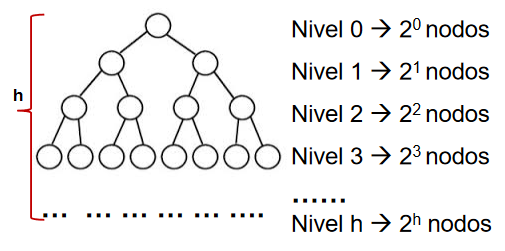
**Ancestro/Descendiente**: Si existe un camino desde n1 a n2, se dice que n1 es ancestro de n2 y n2 descendiente de n1.

**Árbol binario lleno**: Dado un árbol binario T de altura h, diremos que T es lleno si cada nodo interno tiene grado 2 (hijos) y todas las hojas están al mismo nivel (altura).

Si cada nodo tiene 2 hijos y estos están todos a la misma altura (por nivel).

T es lleno si

* T es un nodo simple (árbol binario lleno de altura cero).
* T es de una altura h y sus sub-árboles son llenos de altura h-1.

Cantidad de nodos en un árbol binario lleno es 2h+1-1. 

**Árbol binario completo**: Cuando el árbol es lleno de izquierda a derecha.

Para que un árbol binario sea completo, debe tener todos sus niveles completamente llenos, excepto posiblemente el último nivel, que debe llenarse de izquierda a derecha.

Árbol de decisión: Herramienta de soporte a la toma de decisión que usa un modelo similar a un árbol donde se registran decisiones y sus posibles consecuencias.

Son utilizados en investigación operativa para identificar la mejor estratégia para lograr un objetivo. Se usan en análisis financiero, ocurrencia de eventos, machine learning.

Cada nodo tiene

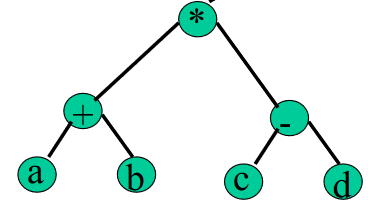
* Información propia del nodo
* Referencia a su hijo izquierdo
* Referencia a su hijo derecho

**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

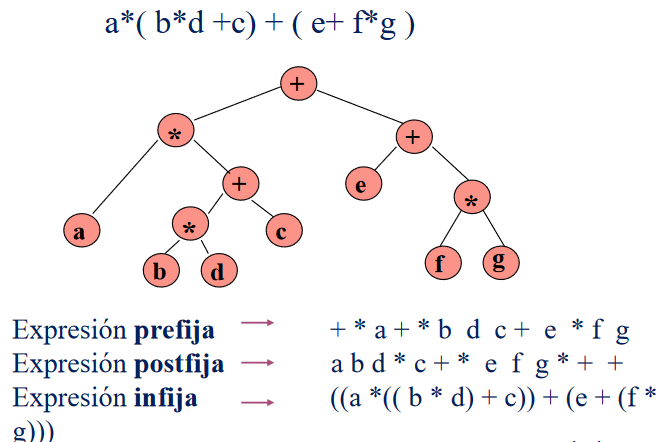
**Por** **niveles**: Se procesan los nodos teniendo en cuenta sus niveles, primero la raíz, luego los hijos, los hijos de éstos, etc.

**Árbol de expresión**: Es un ábol binario asociado a una expresión aritmetica.

* Nodos internos representan operadores
* Nodos externos (hojas) representan operandos.

Usos:

* En compiladores para analizar, optimizar y traducir programas
* Evaluar expresiones algebraicas o lógicas
* No se necesita el uso de paréntesis
* Traducir expresiones a notación sufija, prefija e infija



Preorden: vertice izqquierda dererecha

Postorden: izquierda derecha vertice

Inorden: izquierda vertice derecha

La **cola de prioridad** es una estructura de datos que permite:

* Insert: Insertar un elemento en la estructura, luego se hace un filtrado hacia arriba desde la posición insertada, termina al alcanzar la raíz o un nodo que tenga una clave mayor/menor para restaurar la propiedad de orden. (percolate\_up)
* DeleteMin: Encuentra, recupera y elimina el elemento minimo, luego hace un filtrado hacia abajo desde la raíz para restaurar la propiedad de orden. (percolate\_down)

Se puede implementar en:

* Lista ordenada
  + Insert tiene O(n) operaciones
  + DeleteMin tiene O(1) operaciones
* Lista no ordenada
  + Insert tiene O(1) operaciones
  + DeleteMin tiene O(n) operaciones
* Árbol binario de busqueda
  + Insert y deleteMin tienen O(log n) operaciones

**Heap binaria** es una implementacion de colas de prioridad que no usa punteros y permite implementar ambas operaciones con O(log n) en el peor de los casos.

Cumple con la **propiedad estructural** de árbol binario completo.

Dado que un árbol binario completo es una estructura de datos regular, puede almacenarse en un arreglo.

* La raiz está almacenada en la posición 1.
* El hijo izquierdo está en la posición 2\*i
* El hijo derecho está en la posición 2\*i + 1
* El padre está en la posición i/2

**Propiedad de orden**:

**MinHeap** el elemento mínimo está almacenado en la raiz. El dato almacenado en cada nodo es menor o igual al de sus hijos.

**MaxHeap** se usa la propiedad inversa.

Una heap H consta de

* Un arreglo que contiene los datos
* Un valor que me indica el numero de elementos almacenado

No se usan punteros, facil implementacion de las operaciones.

**DecreaseKey**(x, Δ, H):  
Decrementa la clave que está en la posición x de la heap H, en una cantidad Δ.

**IncreaseKey**(x, Δ, H):

Incrementa la clave que está en la posición x de la heap H, en una cantidad Δ.

**DeleteKey**(x):

Elimina la clave que está en la posición x.

Para construir una heap a partir de una lista de n elementos:

* Se pueden insertar los elementos de a uno (operación insert utilizada en un for)
* se realizan (n log n) operaciones en total

Se puede usar un algoritmo de orden lineal, es decir, proporcional a los n elementos

**BuildHeap**

* Insertar los elementos desordenados en un árbol binario completo
* Filtra hacia abajo cada uno de elementos desde la posición tamaño/2, ya que el resto de los nodos son hojas (la mitad del arbol siempre son hojas). (desde la mitad hasta el inicio) En el filtrado de cada nodo, recorro la altura hacia abajo.

En un árbol binario lleno de altura h que contiene 2 h+1 – 1 nodos, la suma de las alturas de los nodos es: 2 (h+1) – 1 – (h + 1)

Un árbol binario completo tiene entre 2 h y 2h+1 – 1 nodos, el teorema implica que esta suma es de O(n) donde n es el número de nodos.

2 (h+1) – 1 – (h + 1) = n \* log2 n (Lineal) = O(n)

Un **BuildHeap** te sirve para **crear una Heap a partir de un arreglo**, lo malo es que ocupa mucho espacio ya que genera un nuevo arreglo.

**HeapSort**: Solamente ordena una heap

O(n log n)

Hace en bucle lo siguiente:

Armar una Max (creciente, de menor a mayor) o Min (decreciente, de mayor a menor) heap, filtrando hacia abajo todos los elementos de la heap actual (propiedad del orden).

Toma el primer elemento de la Max/Min Heap armada anteriormente y lo intercambia con la última posición de la heap (lo envía al final de la heap y reduce el tamaño de la heap, pero no del arreglo, como un Delete. Esto se hace para que lo que quede al final sea una nueva heap).

Si usas una MaxHeap en un HeapSort, te queda el arreglo de forma creciente (1 -> 2 -> 3)

Si usas una MinHeap en un HeapSort, te queda el arreglo de forma decreciente (50 -> 40 -> 30)