

OBJETO

refiere a:

- objetos del problema: entidades identificadas durante el desarrollo de requerimientos o diseño
- objetos de software: cada representación y modelo en ejecución a una entidad del problema

todos los objetos que son instancias de una clase van a estar caracterizados por los mismos atributos

— Por DISEÑO — (sist software)

una CLASE es un patrón q establece los atributos y el comportamiento de un conj de obj

— Por IMPLEMENTACIÓN de un sist de software

CLASE: es un módulo de software q puede construirse, verificarse y depurarse con cierta independencia de los demás

Desde el pov estático un sist de software orientado a obj es una colección de clases relacionadas.

Desde el pov dinámico un sist de software orientado a obj es un conjunto de obj comunicándose

Cada objeto brinda el conjunto de servicios que define su clase.

objetos se comunican con mensajes y cada uno responde a un msj de acuerdo a su comportamiento.

Un atributo es una propiedad o cualidad relevante q caracteriza a todos los objetos de una clase

Atributo de CLASE: el valor es compartido por todos los obj que son instancias de la clase
representa valores constantes

Atributo de INSTANCIA: los valores varían en cada objeto

SERVICIOS: es una operación q todas las instancias de una clase pueden realizar

→ pueden ser MÉTODOS o CONSTRUCTORES

MÉTODOS

Tienen una funcionalidad q establece su propósito

COMANDOS: al ejecutarse modifica el valor de uno o más atributos del obj q recibió el msj (modifica estado interno) pueden retornar valores.

CONSULTAS: al ejecutarse no modifica el valor de ningún atributo retornan valores. No modifica ESTADO INTERNO del obj

CONSTRUCTOR se invoca al crear el objeto, mismo nombre de la clase inicializa atributos de instancia

sobrecargado: pueden haber varios con \neq número o tipo de parámetros
Si no se define el compilador crea uno automáticamente, los atributos son inicializados por omisión

Los miembros de una clase son atributos y servicios

clase cliente no puede acceder a los atributos de instancia (Privado)
pero puede cambiarlos mediante servicios (comandos)

REQUIERE: es responsabilidad del cliente (ej: tester)

los atributos se definen a través de la declaración de variables

TIPOS de VARIABLES

TIPO **ELEMENTAL**: det un conj de valores y un conj de operaciones que se aplican sobre estos valores.

en ejecución mantiene un valor q corresponde al tipo y participa en las operaciones
conversión implícita entre tipos elementales permite escribir expresiones mixtas del tipo
con operandos de distinto tipo.

TIPO **CLASE** objeto, valor es una referencia (puede ser definida, nula)
ej String ^{Puede ser} indefinida, nula o estar ligada a un obj ^{ligado}

ERROR { de APLICACIÓN compila, ejecuta pero el resultado no es el esperado
de COMPILACIÓN error de sintaxis
de EJECUCIÓN compila, ~~no~~ terminación anormal del sistema
↳ casos de prueba los previenen

un tipo de datos establece un conj de valores y un conj de operaciones que se aplican sobre estos valores

Cuando una clase define un tipo de datos, el conjunto de valores queda determinado por los valores de los atributos, el conj de operaciones lo definen los servicios provistos por la clase. Una variable declarada de un tipo definido por una clase no mantiene un valor dentro del tipo, sino una referencia a un objeto cuyo estado interno mantiene un valor del tipo.

ALCANCE de una variable det su visibilidad → el segmento del código en el cual puede ser usada

una variable declarada en una clase, sea atributo de clase o instancia es visible en toda la clase. Si se declara como privada, solo puede ser usada dentro de la clase completa.

Una variable declarada local a un bloque, se crea en el momento que se ejecuta la instrucción de declaración y se destruye cuando termina el bloque que corresponde a la declaración. El alcance de una variable local es ent el bloque en el q se declara, de modo q solo es visible en ese bloque

PASAJE POR VALOR en el momento q se inicia la ejecución de un servicio, se reserva un espacio de memoria p los parámetros formales y se inicializan con los valores de los argumentos

Una variable declarada como parámetro formal de un servicio se trata como una variable local que se crea en el momento q comienza la ejecución del servicio y se destruye cuando termina. Se inicializa con el valor del argumento o parámetro real. Cada bloque crea un nuevo ambiente de referenciación, formado por todos los identificadores q pueden ser usados. los operandos de una expresión pueden ser constantes, atributos, variables, locales y parámetros visibles en el ambiente de referenciación en el q aparece la expresión

Cuando un objeto recibe un mensaje, su clase det el método que se va a ejecutar en respuesta a ese mensaje. Cuando un obj recibe un msj, el flujo de control se interrumpe y el control pasa al método que se liga al msj. Al terminar la ejecución del método, el control vuelve a la instrucción q contiene el msj.

PASAJE de PARAMETROS

→ Por Valor:

copia el valor de la llamada en el parametro formal

→ Por copia:

si paso un valor de tipo clase copia una referencia al modelo de otro obj

Identidad = referencia

equivalencia = mismo estado interno

ASOCIACIÓN Y DEPENDENCIA

Asociación: Si una clase define un atributo de instancia de tipo de otra clase cuando el modelo de un obj contiene o puede contener al modelo de otro obj

DEPENDENCIA: relación de entre clases A y B, si A declara un parametro o retorna un resultado de clase B cuando el modelo de un obj usa al modelo de otro obj

- Si hay asociación → hay dependencia
- Si una clase A esta asociada a una clase B, el ambiente de referenciamiento de un metodo definido en la clase A No incluye a los atributos y servicios definidos en B.
 - A no tiene acceso directo a los atributos y servicios de B. Los puede consultar
- Si una clase A está asociada a una clase B y A incluye un metodo p la clase B puede definir un metodo p con el mismo número y tipo de Parametros que el metodo p definido en A.

CLIENTES y PROVEEDORES de SERVICIOS

hay clases ^{q son} clientes de los servicios provistos por otras clases proveedoras una misma clase puede cumplir ambos roles

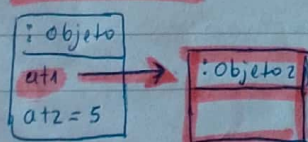
una clase q **usa** los servicios provistos por otra clase, es cliente de la clase q provee dichos servicios. Una clase q **tiene** atributos de otra clase, es cliente de las clases a los que corresponden esos atributos estas

clase **tester** es clase cliente de los servicios q brindan las clases q deben ser verificados

Al implementar una clase **tester** se establece una dependencia entre esta clase y las que van a ser verificados

la clase cliente accede a la clase proveedora a través de su interfaz

Asociación



dependencia

```

Public boolean nombre () {
  objeto2 obj;
  ...
}
  
```

```

Public int nombre (objeto obj) {
  return obj.nombre ();
}
  
```


modifica estado interno

// comandos

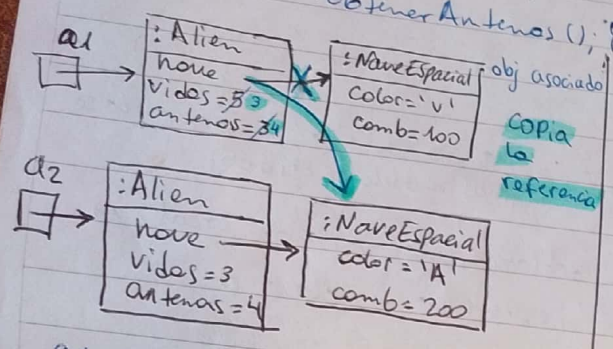
copy (ob : objeto)

COPY

Si obj esta ligado, copia el estado interno del objeto pasado por parametro a la variable ob, en el objeto que recibe el msj. en caso contrario no tiene ningun efecto.

SUPERFICIAL

```
Public void copy (Alien a) {
    nave = a.obtenerNave();
    vidas = a.obtenerVidas();
    antenas = a.obtenerAntenas();
}
```



a1.copy(a2);

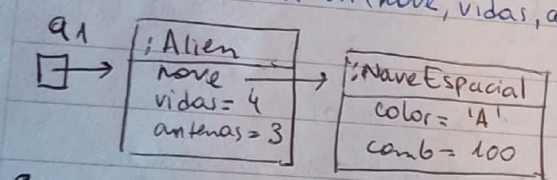
// consultas

CLONE

crea y retorna un nuevo objeto con el mismo estado interno del objeto que recibe el msj.

SUPERFICIAL

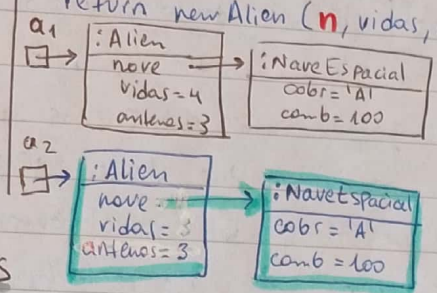
```
Public Alien clone() {
    return new Alien(nave, vidas, antenas);
}
```



a2 = a1.clone(); copian las ref

PROFUNDIDAD

```
Public Alien clone() {
    return new Alien(nave.clone(), vidas, antenas);
}
Public Alien clone() {
    NaveEspacial n = nave.clone();
    return new Alien(n, vidas, antenas);
}
```



las referencias estan ligadas a clones de los obj asociados

EQUALS

Si el obj esta ligado Retorna verdad si el estado interno del obj q recibe el msj es igual al estado interno del obj ligado del parametro.

AMBIENTE de referenciamiento de una instruccion es el conj de identificadores que pueden ser usados o referenciados en esa instruccion; un msj puede incluir 1 o + argumentos o parametros reales (tester) estos deben coincidir con los parametros formales del metodo

EQUALS

Si el obj^{del parametro} esta ligado. Retorna verdad si el estado interno del obj q recibe el msj es igual al estado interno del objeto del parametro.

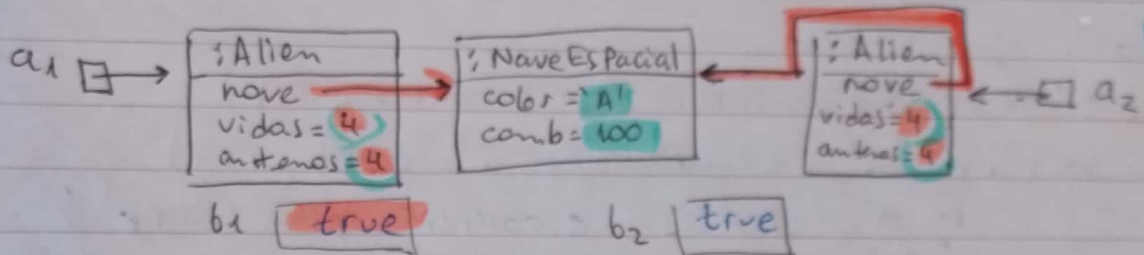
SUPERFICIAL

```
Public boolean equals(Aliena) {
    return nave == a.obtenerNave()
    && vidas == a.obtenerVidas() &&
    antenas == a.obtenerAntenas();
}
```

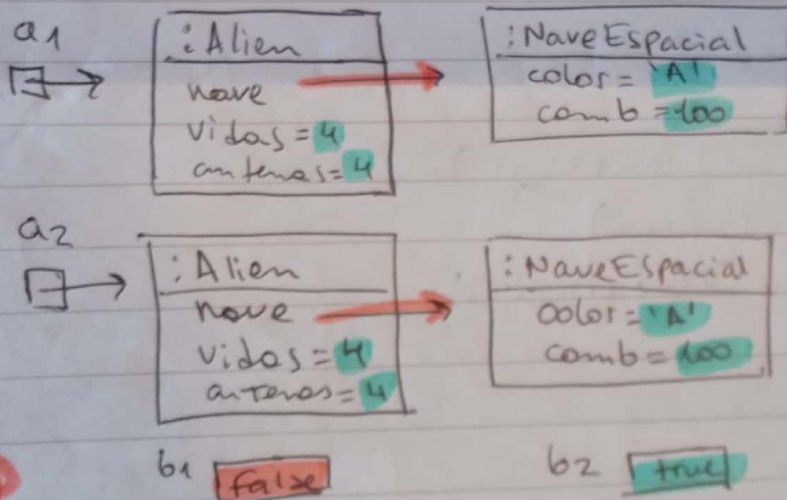
PROFUNDIDAD

```
Public boolean equals(Aliena) {
    return ... vidas == a.obtenerVidas() &&
    nave.equals(a.obtenerNave()) &&
    antenas == a.obtenerAntenas();
}
```

SUP



Prof



Prof

los objetos asociados tienen el mismo estado interno, independientemente de su referencia

SUP

b1 false

b2 true

obj asociados tienen q tener la misma referencia independientemente de su estado interno

Profundidad

copy: se copia el estado interno del objeto de la clase asociada.
 clone: se genera un clone con un nuevo objeto de la clase asociada.
 equals: se compara el estado interno del objeto de la clase asociada.


```
import IPoo.*
```

```
ES. leerChar();  
ES. leerEntero();  
ES. leerEnteroLargo();  
ES. leerFloat();  
ES. leerDoble();
```

1º ver requiere y controlar el cod log no req
2º declarar una variable para el retorno
q es = 0 o null

```
Public boolean equals (TempMinEstacion e) {
```

SUPERFICIAL

```
boolean igual = false;
```

```
if (e != null && contDias() == e.contDias()) {
```

```
    for (int i = 0; i < contDias() && igual; i++)
```

```
        igual = tMin[i] == e.obtenerTempMin(i+1);
```

```
    return igual;
```

obtenerTempMin (dia)
return tMin[dia-1];

Control

```
Public TempMinEstacion clone() {
```

```
    TempMinEstacion e1;
```

```
    e1 = new TempMinEstacion (contDias());
```

```
    for (int i = 0; i < contDias(); i++)
```

```
        e1.establecerTempMin (i+1, tMin[i]);
```

```
    return e1;
```

```
Public void copy (TempMinEstacion e) {
```

```
    if (e != null && contDias() == e.contDias())
```

```
        for (int i = 0; i < contDias(); i++)
```

```
            tMin[i] = e.obtenerTempMin (i+1);
```

```
Public boolean copy (Secuencia Enteros a) {
```

```
    boolean puede = false;
```

```
    if (a != null && contElementos() == a.contElementos()) {
```

```
        Puede = true;
```

```
        for (int i = 0; i <= contElementos(); i++)
```

```
            sec[i] = a.obtenerEntero (i);
```

```
    return puede;
```

```

Public int contFilas() {
    return mot.length;
}
Public int contColumnas() {
    return mot[0].length;
}

```

```

class GrillaAndroides {
    Private Androide [][] grilla;
    Public grillaAndroides (int filas, int columnas) {
        grilla = new Androide[filas][columnas];
    }
}

```

MATRICES

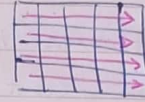
1

Recorrido por fila

```

for (int i=0; i<contFilas(); i++)
    for (int j=0; j<contColumnas(); j++)

```

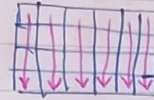


Recorrido por columna

```

for (int j=0; j<contColumnas(); j++)
    for (int i=0; i<contFilas(); i++)

```



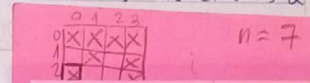
Public boolean **alMenos N** (int n, NaveEspacial unaNave) ^{nigado} $n > 0$

// Retorna true si la grilla contiene al menos n androides asociados a una nave

```

int cont = 0;
for (int i=0; i<contFilas() && cont < n; i++) {
    for (int j=0; j<contColumnas() && cont < n; j++) {
        if (grilla[i][j] != null && grilla[i][j].obtenerNave() == unaNave) {
            cont++;
        }
    }
}
return cont == n;

```



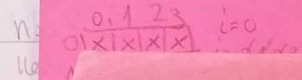
Public boolean **Exactamente N** (int n, NaveEspacial unaNave) ^{nigado} $n > 0$ ^{recorrido} // exhaustiva

// Retorna true si la grilla contiene exactamente n androides asociados a una nave

```

int cont = 0;
for (int i=0; i<contFilas() && cont <= n; i++) {
    for (int j=0; j<contColumnas() && cont <= n; j++) {
        if (grilla[i][j] != null && grilla[i][j].obtenerNave() == unaNave) {
            cont++;
        }
    }
}
return cont == n;

```



// Retorna true si grilla contiene al menos N androides en posiciones consecutivas de una misma fila ^{asociados a una nave} ^{equivalente a una nave}

Public boolean **alMenos N Consecutivos** (int n, NaveEspacial unaNave)

```

int cont = 0;
if (n <= contColumnas()) {
    for (int i=0; i<contFilas() && cont < n; i++) {
        cont = 0; // cambia fila y se resetea cont
        for (int j=0; j<contColumnas() && cont < n; j++) {
            if (grilla[i][j] != null && grilla[i][j].obtenerNave().equals(unaNave)) {
                cont++;
            }
        }
        else {
            cont = 0;
        }
    }
}
return n == cont;

```


Matriz Puntos

f. valido

Public boolean hayNSequidos (int f, int n) { //
 //Retorna true si hay al menos n puntos en posiciones consecutivas con x igual a 0 en la fila f
 int cont = 0;

if (n <= contColumnas())

for (int j = 0; j < contColumnas() && cont < n; j++)

if (mat[f][j] != null && mat[f][j].obtenerX() == 0)
 cont++;

else

cont = 0;

return cont == n; }

Public boolean hayAlMenosNInvEnF (Punto p, int n, int f)

//Retorna true si en la fila f hay al menos n puntos inversos a p

int cont = 0;

for (int j = 0; j < contColumnas() && cont < n; j++)

if (mat[f][j] != null && mat[f][j].inverso(p))
 cont++;

return cont >= n; }

0 <= n <= contColumnas()
 0 <= f <= contFilas()

n = 7
 llega a con = 7

0 <= n
 0 <= f <=

Public boolean hayExactamenteNInversosEnF (Punto p, int n, int f)

//Retorna true si en la fila f hay exactamente n puntos inversos a p.

int cont = 0;

for (int j = 0; j < contColumnas() && cont <= n; j++)

if (mat[f][j] != null && mat[f][j].inverso(p))

cont++;

return cont == n;

Public boolean hayAlSumaNInversosEnF (Punto p, int n, int f)

//Retorna true si en la fila f hay al sumo n puntos inversos a p

int cont = 0;

for (int j = 0; j < contColumnas() && cont <= n; j++)

if (mat[f][j] != null && mat[f][j].inverso(p))

cont++;

return cont <= n; }

8 <= 7 (F)
 6 <= 7 (V)
 7 <= 7 (V)

0 <= n <= contColumnas()
 0 <= f <= contFilas()

ligado

n = 7
 cont

Public boolean hayExactamenteMconN (Punto p, int n, int m) {
 // Retorna true si hay exactamente m filas con al menos n puntos inversos a p

int cont f = 0;

int cont = 0;

for (int i = 0; i < contFilas() && contF <= m; i++) {

cont = 0;

for (int j = 0; j < contColumnas() && cont < n; j++)

if (mat[i][j] != null && mat[i][j].inversa(p))
 cont ++;

if (cont >= n)

cont f ++; }

return cont f == m; }

ligado

$0 \leq n \leq \text{contColumnas}()$
 $0 \leq m \leq \text{contFilas}()$

Public boolean hayAlMenosMconN (Punto p, int n, int m) {

// Retorna true si hay al menos m filas con al menos n puntos inversos a p

int cont f = 0

int cont = 0

for (int i = 0; i < contFilas() && contF <= m; i++) {

cont = 0;

for (int j = 0; j < contColumnas() && cont < n; j++)

if (mat[i][j] != null && mat[i][j].inversa(p))
 cont ++;

if (cont >= n)

cont f ++; }

return cont f >= m; }

g no es una matriz
grilla si es matriz

CLONE

// SUPERFICIAL

```
Public grillaAndroides clone () {
    GrillaAndroides g = new GrillaAndroides (contFilas (), contColumnas ());
    for (int i=0; i < contFilas (); i++)
        for (int j=0; j < contColumnas (); j++)
            if (grilla[i][j] != null)
                g.establecerAndroide (grilla[i][j], i, j);
    return g; }
```

Public grillaAndroides clone () {

// PROFUNDIDAD

```
GrillaAndroides g = new GrillaAndroides (contFilas (), contColumnas ());
for (int i=0; i < contFilas (); i++)
    for (int j=0; j < contColumnas (); j++)
        if (grilla[i][j] != null) // el nulo queda nulo, no se clona
            g.establecerAndroide (grilla[i][j].clone (), i, j);
return g; }
```

COPY

g ligado y de mismo tamaño q grilla

objeto.mensaje()
Verificar si es nulo
antes
de enviar el msg
COPY → verificar
- obj este ligado
- mismas dimensiones
→ equals

```
Public void copy (GrillaAndroides g) {
    if (g != null && contFilas () == g.contFilas () && contColumnas () == g.contColumnas ()) {
        for (int i=0; i < contFilas (); i++)
            for (int j=0; j < contColumnas (); j++)
                grilla[i][j] = g.obtenerAndroide (i, j);
    }
```

// SUPERFICIAL

Public void copy (GrillaAndroides g) {

// PROFUNDIDAD

```
if (g != null && contFilas () == g.contFilas () && contColumnas () == g.contColumnas ()) {
    for (int i=0; i < contFilas (); i++)
```

```
        for (int j=0; j < contColumnas (); j++)
```

```
            if (grilla[i][j] != null)
```

```
                grilla[i][j] = copy (g.obtenerAndroide (i, j))
```

Androide nula . copy (→ error)

```
            else
```

```
                if (g.obtenerAndroide (i, j) != null)
```

// si: grilla[i][j] == null
y g(i,j) no es nulo, esta lig

```
                grilla[i][j] = new Androide (g.obtenerAndroide (i, j).obtenerNavel());
```

crea un nuevo Androide con las el mismo estado interno
ligado que al Androide(i,j) de g

nulo . copy (ligado)

objeto.mensaje(otroobj)

ver

si en la implementación
de mensaje() : objeto

Pregunta si o esta
ligado.

si pregunta ant NO
PREGUNTA si otroobj
esta ligado

COPY
EQUALS Preg
MISMAS DIMENSIONES

EQUALS

PROFUNDIDAD

```

Public boolean equals (MatrizPuntos m) {
    boolean iguales = contFilas() == m.contFilas() && contColumnas() == m.contColumnas();
    for (int i = 0; i < contFilas() && iguales; i++)
        for (int j = 0; j < contColumnas() && iguales; j++)
            if (mat[i][j] != null && m.obtenerPunto(i, j) != null)
                iguales = mat[i][j].equals(m.obtenerPunto(i, j));
            else
                iguales = mat[i][j] == null && m.obtenerPunto(i, j) == null;
    return iguales;
}

```

MATRIZ

SUPERFICIAL

```

Public boolean equals (MatrizPuntos m) {
    boolean iguales = contFilas() == m.contFilas() && contColumnas() == m.contColumnas();
    for (int i = 0; i < contFilas() && iguales; i++)
        for (int j = 0; j < contColumnas() && iguales; j++)
            iguales = mat[i][j] == m.obtenerPunto(i, j);
    return iguales;
}

```

MATRIZ

Public boolean equals (ColeccionRefugios c) } PROFUNDIDAD

```

boolean iguales = true;
if (tamano Coleccion () != c.tamano() || contRefugios() != c.contRefugios())
    iguales = false;
else
    for (int i = 0; i < contRefugios() && iguales; i++)
        iguales = col[i].equals(c.obtenerRefugio(i));
return iguales;

```

mismo cont ligados y no ligados

TABLA

Public void copy (PolizasOrdenadas p) {

```

    if (p.contElementos() == contElementos()) {
        for (int i = 0; i < contElementos; i++)
            polOrd[i] = p.obtenerPoliza(i);
    }
}

```

//revisa si tienen = elemento
si tienen mismo cont de elem
en el arreglo los copia


```

Public boolean equals (PolizasOrdeneas p) { // superficial
    boolean iguales = false;
    if (contPolizas() == p.contPolizas() && contElementos() == p.contElementos()) {
        boolean ig = true;
        for (int i=0; i<contPolizas() && ig; i++)
            if (PolOrd[i] != p.obtenerPoliza(i))
                ig = false;
        iguales = ig;
    }
    return iguales;
}

```

PROF

↳ if (!(PolOrd[i].equals(p.obtenerPoliza(i))))

```

Public PlayaSurtidores clone() { // Superficial TABLA
    PlayaSurtidores ps = new PlayaSurtidores (contPosiciones ());
    for (int i=0; i<tabla.length; i++)
        if (tabla[i] != null) // establecer surtidor reg s ligado
            ps.establisherSurtidor (tabla[i], i); // establezca los No nulos
    return ps;
}

```

s p

```

Public Territorio clone() { // PROFUNDIDAD MATRIZ
    Territorio t = new Territorio (contFilas(), contColumnas());
    for (int i=0; i<contFilas(); i++)
        for (int j=0; j<contColumnas(); j++)
            if (tablero[i][j] != null) // puede usar el establecer
                t.establisherRefugio (tablero[i][j], clone(), i, j);
    return t;
}

```

```

Public Sectores clone() // PROFUNDIDAD
    // como tabla.length
    Sectores s = new Sectores (contSectores());
    for (int i=0; i<contSectores(); i++)
        if (tabla[i] != null)
            s.asignar (tabla[i].clone(), i);
    return s;
}

```

TABLA



4

```
Public void eliminar (int pos) {
```

```
    tabla[pos] = null; }
```

```
Public void eliminar (Androide a) { //busqueda por identidad
```

```
    if (a != null) {
```

```
        for (int i=0; i < tabla.length contElementos(); i++)
```

```
            if (tabla[i] == a)
```

```
                tabla[i] = null; }
```

```
Public int contAndroidesAsignados () {
```

```
    int cont = 0;
```

```
    for (int i=0; i < sectores contElementos(); i++)
```

```
        if (tabla[i] != null)
```

```
            cont ++;
```

```
    return cont; }
```

```
Public int contSectoresAndroide (Androide a) {
```

// exhaustivo

// Retorno cantidad de sectores a los que está asignado el androide a

```
    int cont = 0;
```

```
    if (a != null) {
```

```
        for (int i=0; i < contElementos(); i++)
```

```
            if (tabla[i] == a)
```

```
                cont ++
```

```
    return cont; }
```

```
Public boolean todosAsignados () {
```

```
    boolean todos = true;
```

```
    for (int i=0; i < contElementos() && todos; i++)
```

```
        todos = tabla[i] != null; → if (tabla[i] == null)
```

```
    return todos; }
```

```
        todos = false;
```

```
Public boolean estaAndroide (Androide a) {
```

```
    boolean esta = false;
```

```
    for (int i=0; i < contElementos() && !esta; i++)
```

```
        esta = tabla[i] == a; if (tabla[i] == a)
```

```
    return esta; }
```

```
        esta = true;
```

```
Public boolean existePosicion (int pos) {
```

```
    return 0 <= pos && pos < contElementos(); }
```



```

Public Androide androidSector (int pos) {
    Androide a = null;
    if (existeSector (pos)) → if (0 ≤ pos && pos < cont Elementos())
        a = tabla[pos];
    return a; }

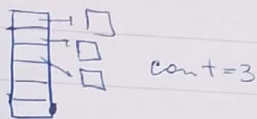
```

```

Public boolean hayANave (Nave Espacial n) { //no exhaustivo
    boolean hay = false;
    if (n != null)
        for (int i = 0; i < cont Elementos() && !hay; i++)
            if (tabla[i] != null && tabla[i].obtenerNave() == n)
                hay = true;
    return hay; }

```

Colección



class HojaEnfermeria

```

Private Signos Vitales [] coleccion;
Private int cont;

```

No ORDENADA

```

Public HojaEnfermeria (int max) {
    coleccion = new Signos Vitales [max];
    cont = 0; }

```

```

Public void insertar (Signos Vitales s) { //Req s ligado if (s != null) {
    coleccion [cont] = s;
    cont++; }
    asigna s a la primer posición libre

```

```

Public boolean estaLlena () {
    return cont == coleccion.length; }

```

```

Public int cantAlarmas () {
    int cont = 0;
    for (int i = 0; i < cont; i++)
        if (coleccion[i].alarma())
            cont++;
    return cont; }

```

```

Private int pos (Signos Vitales s) {
    int toRet = -1;
    for (int i = 0; i < cont && toRet == -1; i++)
        if (coleccion[i].equals(s))
            toRet = i;
    return toRet; }

```

colección No ORDENADA

```

Public void eliminar (Signos Vitales s) {
    //Elimina primera aparición de s, req s ligado, elimina por estado interno
    pone el ultimo objeto ligado en el lugar que queda vacío (pos)

```

```

    int p = pos (s);
    if (p >= 0) {
        coleccion [p] = coleccion [cont-1]; //ultimo elemento ligado de la coleccion
        coleccion [cont-1] = null; //opcional, con insertar se pisa
        cont--; }
    nunca se accede
    recorre hasta cont

```

```

Private int pos (Signos Vitales s) { // Devuelve la posición que tiene
    int toRet = -1; // -1 no encuentra el mismo estado interno que s
    for (int i = 0; i < cont && toRet == -1; i++) // Por identidad
        if (coleccion[i].equals(s)) // if (coleccion[i] == s)
            toRet = i;
}

```

```

return toRet;
}

```

```

Public void eliminar (Signos Vitales s) // 5 ligado

```

```

    int p = pos(s) // Posición de s

```

```

    if (p >= 0) // Posición de s válida (se encuentra s en la coleccion)

```

```

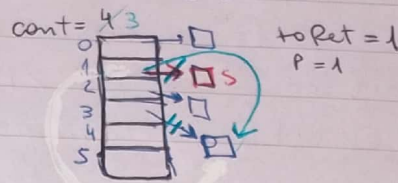
        coleccion[p] = coleccion[cont-1]; // Ult elem ligado de la coleccion

```

```

        coleccion[cont-1] = null; // Opcional, insertar Pissa, recorre hasta cont y no
        cont--; // se accede nunca
    }
}

```



```

Public boolean hayDosAlarmasSeguidas ()

```

```

    boolean hay = false;

```

```

    if (cont > 1) // hay más de un elemento ligado

```

```

        for (int i = 0; i < cont-1 && !hay; i++)

```

```

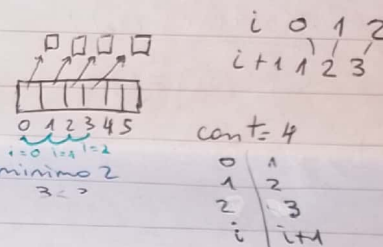
            hay = coleccion[i].alarma && coleccion[i+1].alarma;

```

```

    return hay;
}

```



COLECCIÓN ORDENADA

```

class Coleccion Ciudades Ord

```

```

    Private Ciudad [] col;

```

```

    Private int cont;

```

```

    Public Coleccion Ciudades Ord (int max)

```

```

        col = new Ciudad [max];

```

```

        cont = 0;
    }

```

las ciudades se mantienen ordenadas por cp

Ciudad

mayor (c: ciudad): boolean

Retorna true si el código postal de la ciudad q recibe el mensaje es mayor que el código postal de la ciudad pasada por parametro

// consulta

```

    Public boolean esta lleno ()

```

```

        return cont == col.length;
    }

```

```

    Public boolean Ciudades Unicas ()

```

```

        boolean son = true;

```

```

        for (int i = 0; i < cont-1 && son; i++)

```

```

            if (col[i] == col[i+1])

```

```

                son = false;

```

```

        return son;
    }

```

```

    Public boolean unica (Ciudad c)

```

```

        boolean es = true;

```

```

        for (int i = 0; i < cont && es; i++)

```

```

            if (c == col[i])

```

```

                es = false;

```

```

        return es;
    }

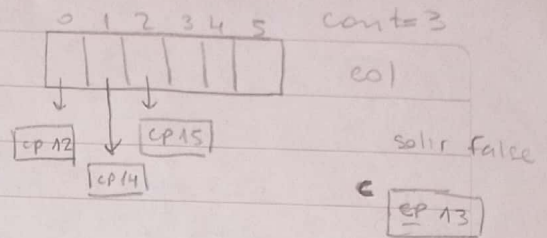
```

c.obtenerCP() == col[i].

obtenerCP()

Reg c ligada

```
private int pos Insercion (Cuidad c) {
    int pos;
    boolean salir = false;
    for (pos = 0; pos < cont && !salir; pos++)
        salir = col[pos].mayor(c);
    return pos; }
```



```
private void arrastrar (int pos) {
    for (int i = cont; i > pos; i--)
        col[i] = col[i-1]; }
```

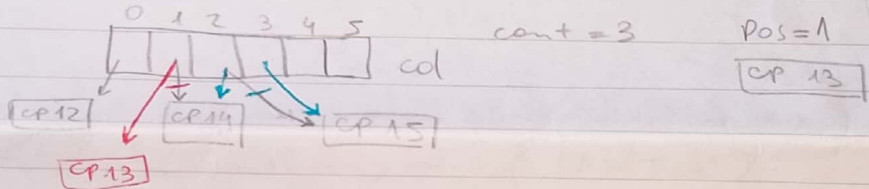
pos = 0
12 > 13 F
pos = 1
14 > 13 V

Public void insertar (Cuidad c) {
//Agrega una nueva ciudad c, considerando la relación de orden establecido por la consulta mayor(cuidad) Reg q colección no este llena, cesteligada

int pos = posInsercion(c);
arrastrar(pos);

col[pos] = c;

cont++; }



```
private int posEliminar (Cuidad c) {
```

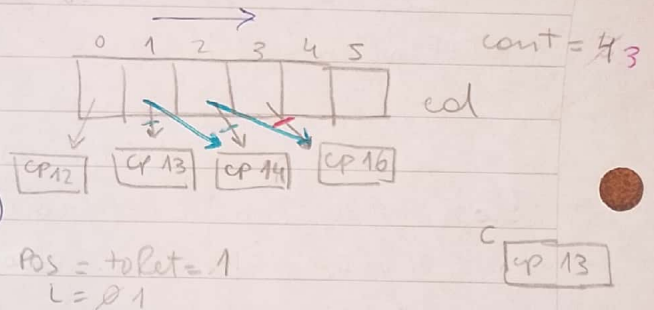
int toRet = -1;

boolean salir = False;

```
for (int i = 0; i < cont && !salir; i++)
```

```
    if (cidades[i].equals(c)) {
        salir = true;
        toRet = i;
    }
```

return toRet; }



```
private void arrastrarArriba (int p) {
```

for (int i = p; i < cont-1; i++)

col[i] = col[i+1]; }

pos = 1
i = 1 1 < 3 V
i = 2 2 < 3 V
i = 3 3 < 3 F

Public void eliminar (Cuidad c) { //Elimina si existe c en la colección y mantiene la relación de orden

int pos = posEliminar(c);

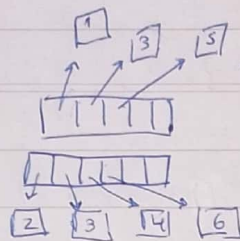
if (pos >= 0) {

arrastrarArriba(pos);

col[cont-1] = null; //opcional

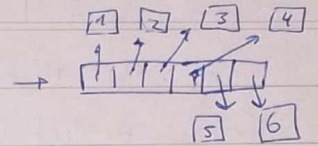
cont--; }

INTERCALAR



cont = 3

recibe msg i
por parametro j



Public PolizasOrdenadas intercalador (PolizasOrdenadas p){

PolizasOrdenadas Pretorno = new PolizasOrdenadas (contPolizas + p.contPolizas());

int $i = 0$; // indice recibe el msg

int $j = 0$; // indice por parametro

while ($i < n$ && $j < p.contPolizas()$)

if ($PolOrd[i].obtenerNroPoliza() < p.obtenerPoliza(j).obtenerNroPoliza()$)

Pretorno.insertar (PolOrd[i]);

$i++$ }

if ($PolOrd[i].obtenerNroPoliza() > p.obtenerPoliza(j).obtenerNroPoliza()$)

Pretorno.insertar (p.obtenerPoliza(j));

$j++$ }

if ($PolOrd[i].obtenerNroPoliza() == p.obtenerPoliza(j).obtenerNroPoliza()$)

Pretorno.insertar (PolOrd[i]);

$i++$;

$j++$ }

if ($i < n$)

while ($i < n$)

Pretorno.insertar (PolOrd[i]);

$i++$; }

if ($j < p.contPolizas()$)

while ($j < p.contPolizas()$)

Pretorno.insertar (p.obtenerPoliza(j));

$j++$; }

return Pretorno;

Pasar de tabla a colección manteniendo el orden

```
private void arrastrarVarios() { //agrupa surtidores
    int escritura = 0;
```

```
    for (int lectura = 0; lectura < cont; lectura++)
```

```
        if (col[lectura] != null) {
```

```
            col[escritura] = col[lectura];
```

```
            escritura++;
```

```
        }
    }
    for (int i = escritura; i < contPosiciones(); i++)
```

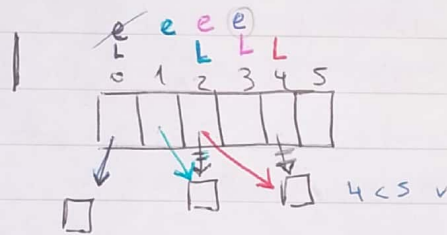
```
        col[i] = null; }
```

```
while (escritura < cont) {
```

```
    col[escritura] = null;
```

```
    escritura++; }
```

contPosiciones = 6
cont = 5



```
public void eliminarDevueltos() {
```

```
    int eliminados = 0;
```

```
    for (int i = 0; i < cont; i++) {
```

```
        if (col[i].estaDevuelto()) {
```

```
            col[i] = null;
```

```
            eliminados++;
```

```
        }
    }
    arrastrarVarios();
```

```
    cont = cont - eliminados; }
```

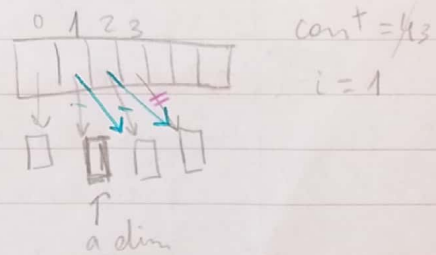
```
private void arrastrar(int pos)
```

```
    for (int j = pos; j < cont; j++)
```

```
        col[j] = col[j+1];
```

```
    col[cont] = null; }
```

3



```
public void eliminarV2 (Prestamo p)
```

```
    boolean encuentre = false;
```

```
    for (int i = 0; i < cont && !encuentre; i++)
```

```
        if (col[i] == p) {
```

```
            encuentre = true;
```

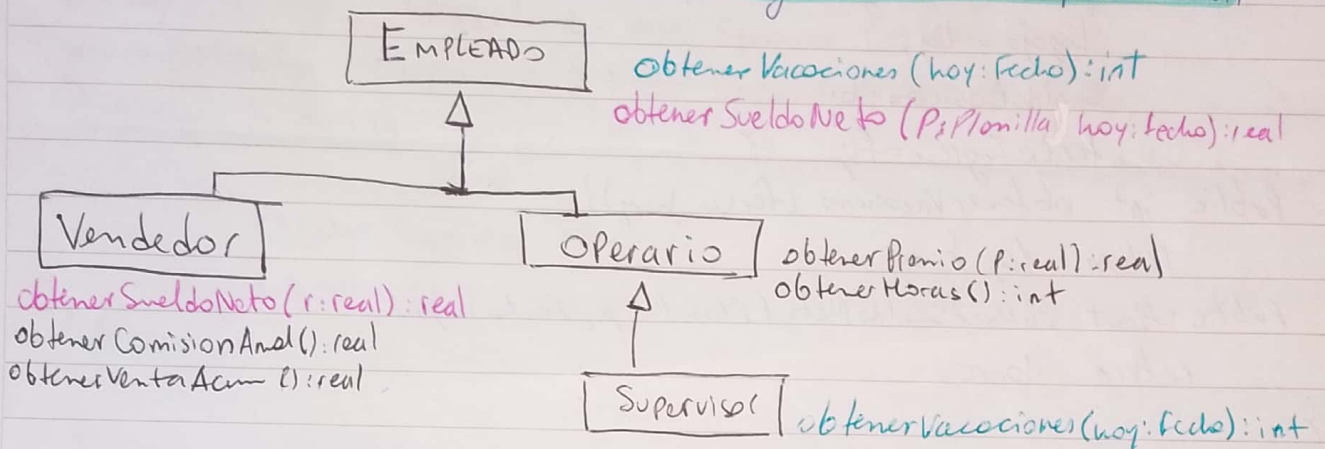
```
            cont--;
```

```
            arrastrar(i); }
```

HERENCIA es una

1

La herencia jerárquica es un mecanismo que permite organizar clases de acuerdo a relaciones de generalización-especialización



Clases Vendedor y Operario son clases derivadas de la clase Empleado
 Todo obj de clase Vendedor es un obj de clase Empleado
 Los constructores no se heredan, los atributos y métodos sí
 Los atributos protegidos pueden ser accedidos desde las clases derivadas

REDEFINICIÓN

Se produce cuando una clase derivada (supervisor) define un método con la misma signatura que un método de una clase ancestro (Empleado)
 cambia la implementación, mismo ^{tipo} retorno, mismos ^{tipos} parámetros,
 mismo nombre.

SOBRECARGA

Cuando dos o más métodos tienen el mismo nombre pero distinto tipo o número de parámetros el identificador → sobrecargado

Empleado e1 = new Empleado (125, 2100, f);

Operario e2 = new Operario (126, 2000, f);

Supervisor e3 = new Supervisor (120, 3000, f);

Vendedor e4 = new Vendedor (129, 5000, f, 10000);

int c ≠ {ejecuta version float c; ejecuta

c = e1.obtenerVacaciones(h); Empleado c = e1.obtenerPremio(15); Error

c = e2.obtenerVacaciones(h); Empleado c = e2.obtenerPremio(15); Operario

c = e3.obtenerVacaciones(h); Supervisor c = e3.obtenerPremio(15); Operario

c = e4.obtenerVacaciones(h); Empleado c = e4.obtenerPremio(15); Error

b1 = e1.obtenerSueldoNeto(p, h); Empleado b4 = e4.obtenerSueldoNeto(p, h); Empleado

b2 = e2.obtenerSueldoNeto(p, h); Empleado b5 = e4.obtenerSueldoNeto(r); Vendedor

b3 = e3.obtenerSueldoNeto(r); Error

float
b1, b2, b3, b4
b5;


```

class Empleado {
    protected int legajo;
    protected float sueldoBasico;
    protected fecha fechaIngreso;
    public Empleado(int leg, float sb, fecha fi) {
        legajo = leg;
        sueldoBasico = sb;
        fechaIngreso = fi;
    }
    public int obtenerVacaciones (fecha hoy) {
        return 10;
    }
    public float obtenerSueldoNeto (Planilla p, fecha hoy) {
        return 10.000;
    }
}

```

```

class Vendedor extends Empleado {
    protected float ventasAc;
    public Vendedor (int leg, float sb, fecha fi, float v) {
        super (leg, sb, fi); // primero en el constructor
        ventasAc = v;
    }
}

```

Las clases especializadas heredan atributos y comportamiento de las clases generales y agregan atributos y comportamiento específico. Una clase especializada puede también redefinir el comportamiento establecido por su clase más general. Dada una clase es posible definir otras más específicas que heredan los atributos y comportamiento de la clase general y agregan atributos y comportamiento especializado.

Herencia es un recurso poderoso porque **favorece la extensibilidad**. Con frecuencia los **cambios** en la especificación del problema se resuelven incorporando nuevas clases especializadas, sin necesidad de modificar las que ya han sido implementadas, verificadas e integradas al sistema. **Facilita la reusabilidad** porque no solo se reutilizan clases, sino colecciones de clases relacionadas a través de herencia.

Herencia SIMPLE clasificación jerárquica (árbol), cada subclase corresponde a una única clase base. Cada clase puede derivar entonces en una o varias subclases o clases derivadas pero solo puede llegar a tener una única clase padre. Las clases descendientes de una clase son las que heredan de ella directa o indirectamente incluyéndole a ella mismo.

EXTENSIBILIDAD: ^{herencia} permite que los cambios en los requerimientos puedan resolverse definiendo nuevas clases, sin modificar las clases que ya han sido desarrollados y verificados.

POLIMORFISMO

mecanismo que permite que un mismo nombre este asociado a distintas abstracciones

Variable Polimorfica puede quedar asociada a obj de diferentes clases

Empleado e1, e2; Empleado e = new Operario (1, 1000, f); (funciona)

gral → + específica Operario o = new Empleado (2, 1000, f); Error en compilación

Asignacion Polimorfica liga un obj de una clase a una variable declarada de otra clase

e = new Supervisor (3, 500, f3);

e = new Operario (4, 600, f4);

Método Polimorfico incluye una o más variables polimorficas como parametro

Planta → Empleado

coleccion polimorfica (clase polimorfica) pq define métodos polimorficos

Clase polimorfica incluye uno o más métodos polimorficos

vinculaciones

Las ligaduras estáticas son las que se establecen antes de la ejecución

Las ligaduras dinámicas son las que se establecen y pueden cambiar durante la ejecución

La ligadura dinámica de código es la vinculación en ejecución de un mensaje con un método

Tipo ESTÁTICO de una variable es el tipo que aparece en la declaración

Tipo DINÁMICO de una variable se det en ejecución y corresponde a la clase del obj referenciado

Empleado o1 = new Operario (125, 200, f);

Operario o2, o3;

o2 = o1

o3 = new Empleado (126, 200, f);

error de compilación

tipo estático restringe el polimorfismo en las asignaciones

Empleado o1 = new Operario (125, 200, f);

Operario o2 = new Operario (126, 200, f);

float p1 = o1.obtenerPremio (10);

float p2 = o2.obtenerPremio (10);

error de comp

Polimorfismo

Permite que las diferentes entidades de una misma clase puedan exhibir distintas formas de un mismo comportamiento. En POO es el mecanismo que un mismo nombre pueda quedar ligado a objetos de diferentes clases y que objetos de distintas clases puedan recibir un mismo mensaje y cada uno actúe de acuerdo al comportamiento establecido por su clase.

En el desarrollo de un Sist de software la herencia es un recurso importante porque favorece a la productividad porque permite que un mismo nombre pueda asociarse a abstracciones diferentes, dependiendo del contexto

REUSABILIDAD

EXTENSIBILIDAD

Polimorfismo: refiere a la capacidad de asociar diferentes definiciones a un mismo nombre, de modo que el contexto determine cuál corresponde

Ligadura Dinámica de Código: es la vinculación en ejecución de un mensaje con un método.

Cuando un método definido en una clase, queda redefinido en una clase derivada, el tipo dinámico de la variable determina qué método va a ejecutarse en respuesta a un mensaje.

Polimorfismo es un mecanismo que favorece la reusabilidad pero debe restringirse para brindar confiabilidad. Los chequeos de tipo en compilación garantizan que no van a producirse errores de tipo en ejecución.

Java chequeos de tipos establece restricciones sobre

- asignaciones polimórficas
- los mensajes que un objeto puede recibir

Los mecanismos de herencia, polimorfismo y vinculación dinámica favorecen la extensibilidad porque reducen el impacto de los cambios. Si una estructura de datos está conformada por objetos que pertenecen a una colección de clases, la colección puede incorporar a nuevas clases, sin afectar a la estructura de datos.

Polimorfismo, redefinición de métodos y ligadura dinámica de código la posibilidad de que una variable pueda referenciar a objetos de diferentes clases y que un método pueda ser redefinido en las clases derivadas, brinda flexibilidad al lenguaje, siempre que además exista ligadura dinámica de código.

Principio fundamental del paradigma de prog orientado a objetos es construir un sistema de software con base en las entidades de un modelo elaborado a partir de un proceso de

abstracción y clasificación

Modelo computacional de la POO es un conj de objetos (de software) comunicándose a través de mensajes. Cada objeto responde a un mensaje de acuerdo al comportamiento determinado por su clase

3

DISEÑO ORIENTADO a objetos

Consiste en definir una colección de clases relacionados entre sí. Las clases se relacionan entre sí a través de

ASOCIACIÓN (tiene un), atributo de instancia de una clase, corresponde a otra clase.

DEPENDENCIA (use un) los métodos de una clase reciben parámetros o declaran variables locales de otra clase.

HERENCIA (es un) las instancias de una clase derivada son también instancias de las clases de las cuales hereda.

abstracción cuando agrupamos objetos en clases,

superabstracción: proceso de clasificar clases. Parte de una clase muy gral y descomponiéndola en otras más específicas identificando diferencias entre los objetos. Si el proceso continúa hasta alcanzar subclases homogéneas → especialización

Agrupando objetos de un conjunto en clases según sus atributos y comportamiento. Estas clases serán a su vez agrupadas en otras de mayor nivel hasta alcanzar la clase más gral → generalización

CLASE ABSTRACTA

fue creada para lograr un modelo más adecuado. En ejecución no va a haber objetos de software de una clase abstracta

→ Puede incluir uno, varios, todos o ninguno métodos abstractos. Un método abstracto no puede implementarse de manera gral para todas las instancias de la clase. Define solo la signatura. La implementación concreta no puede generalizarse.

→ Si una clase hereda de una clase abstracta y no implementa todos los métodos abstractos también debe ser definida como abstracta. El constructor de la clase abstracta solo va a ser invocado desde los constructores de las clases derivadas. Una clase concreta debe implementar todos los métodos abstractos de sus clases ancestro.

Tipo de dato abstracto: es un patrón a partir del cual es posible crear instancias sin conocer la representación interna de los valores ni la implementación de las operaciones.
ej: float, String

GENERICIDAD

→ abstraer lo que es común a un conjunto de entidades para definir un concepto que los incluya a todos GENERALIZACIÓN

→ es recurso poderoso porque favorece la extensibilidad y la reusabilidad. La **extensibilidad** reduce el impacto de los cambios los modificaciones con frecuencia pueden resolverse definiendo nuevas clases específicas, sin necesidad de cambiar los que ya han sido desarrollados y verificados. La **reusabilidad** evita escribir el mismo código repetidamente acelerando el proceso de desarrollo.

POO tiene como obj pnal favorecer la confiabilidad, reusabilidad y extensibilidad del software

→ enfoque: • en el **diseño** reduce la complejidad en base a la **descomposición** del problema en piezas más simples, a partir de un conjunto de clases relacionadas entre sí.

• En la etapa de **implementación** utilizar un lenguaje que permita **retener** las relaciones entre las clases y **encapsular** su representación interna.

La **abstracción** de datos y el **encapsulamiento** permiten que una o más clases usen los servicios provistos por otra clase considerando sólo su comportamiento, sin tener en cuenta como lo implementa.

La **herencia** permite aumentar el nivel de abstracción mediante un proceso de clasificación en niveles.

Una **clase genérica** es aquella que **encapsula** a una estructura cuyo comportamiento es independiente del tipo de sus elementos.

una clase genérica brinda un conjunto de servicios generales cuya implementación es independiente del tipo de los componentes de la estructura.

La clase genérica puede ser usada para modelar diferentes aplicaciones.

En muchas aplicaciones es posible reusar una clase genérica, pero es necesario extenderla con otra que brinde el comportamiento específico.

objeto evento	Interface de Oyente	Manejador
† ActionEvent	Action Listener	actionPerformed (Action Event)