Lenguaje Java

Características de Java
Herencia, Polimorfismo...

José Manuel Pérez Lobato

Características de Java

- Herencia
- Polimorfismo
- Interfaces
- Clases abstractas
- Modificadores y visibilidad
- Inner Class

Herencia

- Es un **mecanismo** utilizado **para generalizar** determinadas clases de objetos de manera que se pueden definir particularizaciones de una clase determinada (Java no admite herencia múltiple).
- Sirve para representar las **relaciones** entre clases de tipo "**ES UN**"
- Permita **organizar las clases en una jerarquía** en la que las clases hijas (subclases) heredan los atributos y métodos de las clases padres (superclases) y además pueden introducir atributos y métodos propios.
- La herencia **es transitiva**. A puede heredar de B, y B puede heredar de C. En ese caso A tendrá los métodos definidos en B y en C (los de C no redefinidos en B)
- **Ventajas**: Ahorra trabajo y facilita el mantenimiento (cambios en 1 clase se ven reflejados en todas sus herederas, por ejemplo cambio del cálculo de la letra del dni para personas cambiará el método para alumnos, profesores, empleados, y demás clases que hereden de persona)

Herencia

- Sintaxis en java:

  class Clase*Hija* **extends** *ClasePadre* {

  . . . . .

  }

- **Ejem.**: Definir una clase "*FiguraBidimensional*" como una clase genérica de las clases "*Triángulo*", "*Cuadrado*" y "*Círculo*", donde Cuadrado también será un particularización de la clase Cuadrilatero
- Ejem: Para definir la relación mencionada anteriormente se debería escribir:

```
class FiguraBidimensional{...
}
class Cuadrilatero extends FiguraBidimensional{...
}
class Cuadrado extends Cuadrilatero{...
}
```

Los objetos de la clase *Cuadrado* tendrán las propiedades y métodos definidos por la clase *FiguraBidimensional y por la clase Cuadrilatero*

Herencia:super

- Si se quiere hacer uso de los métodos y propiedades del padre de la clase y existe ambigüedad se deberá usar la palabra reservada ***super*** seguida de un punto y del método o propiedad que se quiere invocar o conocer o modificar su valor. También se usa super para invocar al constructor del padre.

```
public class Triangulo {
 double lado1,lado2,lado3;
   Triangulo ( double l1,double l2, double l3){
     lado1=l1;
    lado2=l2;
    lado3=l3;
   }
   double perimetro(){
     return (lado1+lado2+lado3);
   }
   double area(){  // si no es un Δ devolverá NaN
      double s=(lado1+lado2+lado3)/2;
      return (Math.sqrt(s*(s-lado1)*(s-lado2)*(s-lado3)) );
   }
}
```

```
public class TrianguloEquilatero extends Triangulo {
 TrianguloEquilatero(double l ){
   super ( l, l, l );
 }
 double area(){
   return (Math.sqrt(3)*lado1*lado1/4);
 }
 double area2(){ return super.area(); }
 public static void main(String[] args) {
   TrianguloEquilatero t= new TrianguloEquilatero(3);
   System.out.println(t.perimetro());
   System.out.println(t.area());  //la de TrianguloEquilatero
}
}
```

Constructores de clases extendidas

- En el constructor de la subclase (=clase hijo, =clase extendida) se puede invocar uno de los constructores de la superclase (=clase padre, =clase base)
- La invocación al constructor de la clase padre con: **`super(…)`**
- La **llamada al constructor** de la clase **padre debe ser la primera sentencia del constructor** de la clase extendida, excepto si se llama a otro constructor de la misma clase utilizando this(…)

```
public class Triangulo implements Cloneable{
double lado1, lado2, lado3;

Triangulo(double l1, double l2, double l3) {
  lado1 = l1;
  lado2 = l2;
  lado3 = l3;
}
```

```
Triangulo(int l){
  this(l,l,l);
}
Triangulo (){
  super();
  lado1=lado2=lado3=1;
}
```

Constructores de clases extendidas

- Si **no se invoca explícitamente ningún constructor de la superclase**, implícitamente **se llama al constructor sin argumentos** de la superclase.
- Java proporciona **un constructor no-arg por defecto**, que llama al constructor no-arg de la superclase
- **Cuando se crea un objeto de la clase hijo se crea otro de la clase padre** (Ejercicio: comprobarlo colocando un atributo en la clase base que cuente cuantos objetos de la misma se crean.)

Anulación de métodos

- Un objeto de la clase hijo puede usar los atributos, y los métodos de la clase padre sin necesidad de redeclararlos.
- La clase hijo puede definir métodos propios.
- La clase hijo puede sustituir un método de la clase padre por uno propio, pero:
  - Si el método en el padre es estático, debe serlo también en el hijo.
  - Los nombres de los métodos y los tipos de los parámetros deben ser idénticos y deben devolver el mismo tipo de valor

Referencias ClaseBase--ClaseExtendida

- **Se puede** hacer una asignación de una referencia de objeto de la clase padre a un obj. de la clase hijo

```
ClasePadre hijo= new ClaseHijo();
hijo.metodo()
// Persona p = new Alumno ("Juan");
//p.método();
```

- No se puede hacer una asignación de una referencia de objeto de la clase hijo a una referencia de la clase padre, aunque si se hace una conversión explícita y la referencia apunta realmente a un objeto de la clase hijo sería correcto
  - Sería **erróneo** (en compilación)

```
ClasePadre sup = new ClasePadre();
ClaseHijo ext = sup;
```

  - También da **error** en ejecución (no en compilación):

```
ClaseHijo ext=(ClaseHijo)sup;
```

si `sup` no apunta a un objeto de la clase `ClaseHijo`

Uso Métodos y Atributos con referencias de ClasePadre

```
ClasePadre hijo= new ClaseHijo();
hijo.metodo(); //el de ClaseHijo
hijo.atributo; // el de ClasePadre
```

- **Cuando se invoca a un método (no estático) se ejecuta el que se corresponda con la clase del objeto** no con la clase de la referencia.
- Por el contrario, **cuando se utiliza un atributo se tiene en cuenta el tipo de la referencia** no el del objeto real.
- En compilación se verifica la llamada por el tipo de la referencia por lo que el `metodo()` debe existir en la clase padre para evitar error de compilación, si también existe, redenominado en el hijo, se ejecutará el del hijo
- Si el `metodo()` sólo existe en el hijo tendré que hacer conversión explícita : `((ClaseHijo)hijo).metodo()` para evitar el error de compilación

instanceof

- El operador *instanceof* devuelve true si **el objeto** es de la clase indicada o la clase del objeto es heredera de la indicada.
- Uso:

```
obj1 instanceof Clase
```

```
ClasePadre ext = new ClaseHijo();
ClasePadre sup = ext;
ClasePadre sup2=new ClasePadre();

ext instanceof ClasePadre    => true
ext instanceof ClaseHijo   => true
sup instanceof ClasePadre    => true
sup instanceof ClaseHijo   => true
sup2 instanceof ClaseHijo  => false
```

NOTA: Existen otras formas de conocer la clase de un objeto por ejemplo
Objeto.getClass().toString() devuelve el nombre de la clase

Ocultamiento de campos

- Se declara un atributo en la clase hijo con el mismo nombre que el de la clase padre.
- Los dos atributos existen. Para acceder al de la clase padre desde la clase hijo se usa: `super`

- Cuando se accede a un atributo, se usa la clase de la referencia para saber a cuál nos referimos

```
ClasePadre ext = new ClaseHijo();
ClasePadre sup = ext;
System.out.println("sup.str= "+sup.str); //SuperStr
System.out.println("ext.str= "+ext.str); //SuperStr
```

Ejemplo Herencia

```
class ClasePadre {
  public String str ="SuperStr";
  public void show () {
    System.out.println("ClasePadre.str: "+str);
   }
  }
class ClaseHijo extends ClasePadre {
  public String str ="ExtendStr";
  public void show () {
    System.out.println("ClaseHijo.str: "+str);
    System.out.println ("ClaseHijo.super.str:"+super.str);
    super.show();
  }
public static void main (String[]args) {
  ClaseHijo h= new ClaseHijo();
  ClasePadre p= ext;
  p.show(); // El show de la clase Hijo (el del tipo del objeto)
  h.show(); // También el show de la clase Hijo
  System.out.println ("p.str= "+p.str);  //str de Padre
  System.out.println ("h.str= "+h.str);  //str de Hijo
  }
}
```

Ejemplo Herencia (II)

- Hay sólo 1 objeto; dos referencias a él, una como clase hijo (`h`) y otra como clase padre (`p`)
- El método `show` que se ejecuta desde el main es siempre de la clase extendida
- Al usar `p` se usa el atributos `str` de la clase padre y al usar `h` el de la clase hijo.
- Resultado:

```
ClaseHijo.str: ExtendStr
ClaseHijo.super.str:SuperStr
ClasePadre.str: SuperStr

ClaseHijo.str: ExtendStr
ClaseHijo.super.str:SuperStr
ClasePadre.str: SuperStr

p.str= SuperStr
h.str= ExtendStr
```

Ejemplo Herencia (III): Exception

- La clase Exception, al igual que las otras clases (no *final),* puede ser heredada
- Al heredar de la clase Exception creamos una nueva clase Exception propia que podemos utilizar como deseemos.

```
void consultarAlumno (int matricula) throws AlumnoNoExiste {
    int pos=-1;
    try {
        pos=buscarAlumno(matricula);
        listaAlumnos[pos].mostrar();
    } catch (AlumnoNoExiste  e) {
        System.out.println ( e.toString() );
    }

    }
int buscarAlumno(int matricula) throws AlumnoNoExiste  {
    int pos=-1;
    for (int i=0; i<numAlumMatriculados && pos==-1 ;i++)
        if (listaAlumnos[i]!= null && listaAlumnos[i].getNumMatricula()==mat)
                pos=i;
    if (pos >=0)     return pos;
        else throw new AlumnoNoExiste (matricula);
}
```

```
public class AlumnoNoExiste extends Exception {
        AlumnoNoExiste  (int  matricula) {
          super("No existe el alumno:" + matricula);
        }
        . . .
    }
```

Clase Object

- Una clase que no extiende explícitamente de otra clase extiende implícitamente la clase Object de Java

- Todas las clases extienden de Object (java.lang.Object), directa o indirectamente, y por tanto heredan sus métodos (equals, hashCode, clone, toString,...). Aunque hay que definirlos explícitamente en nuestra clase si queremos que funcionen correctamente con ella.

- Si se desea una referencia para almacenar cualquier tipo de objeto se declarará de tipo Object.

```
class CObject{
Object v[];
  CObject (int n, char c) {
    v=new Object[2];
    v[0] = new Integer(n);
    v[1]= new Character(c);
  }
  void mostrar (){
for (int i=0; i<v.length; i++) {
     if (v[i] instanceof Integer)
       //Conversión explicita porque intValue() no existe en la clase Object
        System.out.println (((Integer)v[i]).intValue());
     if (v[i] instanceof Character)
        System.out.println (((Character)v[i]).charValue());
    }
  }
  void mostrar2() {
    for (int i=0; i<v.length; i++)
    System.out.println (v[i]);
  }
```

```
public static void main (String a[]) {
  CObject obj=new CObject(33, 'A');
  obj.mostrar();
  obj.mostrar2();
}
```

Método clone

- Sirve para duplicar un objeto
- Las clases que lo codifiquen tienen que implementar el interfaz Cloneable que no tienen ningún método:

  public interface Cloneable { }
- Si se añade el método clone() a una clase sin implementar el interface Cloneable se producirá la excepción CloneNotSupportedException

Método clone- Ejemplo

```
public class Triangulo implements Cloneable{
  double lado1, lado2, lado3;

  Triangulo(double l1, double l2, double l3) {
    lado1 = l1;
    lado2 = l2;
    lado3 = l3;
  }
  double perimetro() {
    return (lado1 + lado2 + lado3);
  }
@Override
  protected Object clone() throws
       CloneNotSupportedException {
    return super.clone();
  }
```

```
public static void main(String[] args) {
  Triangulo t= new Triangulo (3,3,3);
  Triangulo t2=null;
  try {
    t2=(Triangulo)t.clone();
  }catch (CloneNotSupportedException e) {
     System.out.println("Error");
     e.printStackTrace();
  }
  t2.lado1=4;
  System.out.println("perimetro
t2:"+t2.perimetro());
  System.out.println("perimetro
t:"+t.perimetro());
  }
}
```

```
public class Persona implements Cloneable{
  String nombre;
  int edad;
  @Override
  protected Object clone() throws CloneNotSupportedException {
     return super.clone();
  }
  public Persona(String nombre, int edad) {
super();
this.nombre = nombre;
this.edad = edad;
  }
@Override
public String toString() {
return "Persona [nombre=" + nombre + ", edad=" + edad + "]";
}
}
```

```
public static void main(String[] args) throws
CloneNotSupportedException {
Persona p= new Persona("Juan", 11);
Persona p2= p;
Persona p3= (Persona) p.clone();
p.nombre="otro nombre";
System.out.println(p);
System.out.println(p2);
System.out.println(p3);
}
```

Resultado ejecución

```
Persona [nombre=otro nombre, edad=11]
Persona [nombre=otro nombre, edad=11]
Persona [nombre=Juan, edad=11]
```

Polimorfismo

- Es una característica que permite que **métodos con el mismo nombre puedan efectuar acciones distintas**. Esto se puede conseguir de varias formas:
  - Colocando al **método parámetros de distinto tipo**, con lo cual tendrá varias implementaciones.
  - Colocando como **parámetro** una **referencia a un objeto de una clase base** B, lo que permitirá invocar al método con cualquier objeto de la clase B o de una de sus subclases
  - Hacer lo mismo que en el apartado anterior, pero utilizando un **parámetro de una interface** con lo que cualquier objeto de una clase que implemente dicha interfaz podrá ser recibido como parámetro.
  - **No existe polimorfismo** si la única diferencia es el tipo de **valor devuelto. Es un error.**

- Existe un polimorfismo dinámico (ligadura dinámica) cuando es en tiempo de ejecución cuando se decide qué método hay que utilizar.

- También existe un polimorfismo de clases cuando creamos una clase B como extensión de otra A, ya que un objeto de la clase B podría ser utilizado también como objeto de la clase A.

Ejemplo Polimorfismo (I)

```
class Cuadrado{
  int lado;
  int area(){
       return lado*lado;
  }
}
class Triangulo{
  int base, altura;
  int area(){
       return (base*altura)/2;
  }
}
void calculoArea(Cuadrado c, Triangulo t){
  System.out.println("El área del cuadrado es:"+c.area());
  System.out.println("El área del triángulo es:"+t.area());
  }
```

Ejemplo Polimorfismo (II)

```
class SobreCarga{
  int a, b;
  SobreCarga(){
    a=0;
    b=0;  }
  SobreCarga(int n){
    a=n;
    b=0;  }
  SobreCarga(int n, int m){
    a= n;
    b= m;  }
  void incrementar(char c){
    a=a+(int) c;
    b=b+(int) c;  }
  void incrementar(int c){
    a=a+c;
    b=b+c;  }
  void escribirAtributos(){
    System.out.println(a+"
   "+b);
  }
}
```

```
class CSobreCarga{
public static void main (String a[]) {
  SobreCarga s1= new SobreCarga();
  SobreCarga s2= new SobreCarga(2);
  SobreCarga s3= new SobreCarga(2,3);

   s1.incrementar((char)1);//s1.increment
   ar('A');
  s2.incrementar(5);
  s1.escribirAtributos(); // 1 1
  s2.escribirAtributos(); // 7 5
  s3.escribirAtributos(); // 2 3
}
}
```

Ejem. Polimorfismo (IIIa)

```
class CuentaBancaria{
  double saldo=0;
  double interes=1.5;
  double comision=10;
  CuentaBancaria (double saldo){
    this.saldo=saldo;
  }
  void anadeInteres(){
    saldo=saldo+ saldo* interes/100 -comision;
  }
}
class CuentaAhorro extends CuentaBancaria {
  CuentaAhorro (double saldo, double interes) {
    super(saldo);
    this.interes=interes;
  }
  void anadeInteres() {
    saldo=saldo + saldo * interes/100;
  }
}
```

Ejem. Polimorfismo (IIIb)

```
class CPolimorf {
void operacionBanca (CuentaBancaria cb){
    cb.anadeInteres();
  }
public static void main (String[]args) {
  CPolimorf cf=new CPolimorf();
  CuentaAhorro ca = new CuentaAhorro(100, 5);
  CuentaBancaria cb = new CuentaBancaria(200);

  System.out.println ("saldo ca="+ ca.saldo); // 100
  System.out.println ("saldo cb="+ cb.saldo); // 200
/* llama a CuentaAhorro.anadeInteres, si CuentaAhorro.anadeInteres
   no existiera llamaria a CuentaBancaria.anadeInteres  */
  cf.operacionBanca (ca);

/* llama a CuentaBancaria.anadeInteres */
  cf.operacionBanca (cb);
  System.out.println ("saldo ca="+ ca.saldo); // 105
  System.out.println ("saldo cb="+ cb.saldo); // 193
  }
}
```

Interfaces hasta java 7

- Son un tipo especial de “clase”, que **no incluye la definición de sus métodos**, consta de una lista de métodos que se pueden incluir en cualquier otra clase.
- Las clases, en las que todos los métodos son abstractos deberían definirse como interfaces.
- Los métodos definidos (implementado su comportamiento) en las interfaces deben ser codificados en las clases que los implementan.
- **Una clase que implemente una interfaz, debe implementar todos sus métodos**. Si sólo implementa algunos debe ser declarada abstracta y no se podrá instanciar (crear objetos de la misma).
- Una **clase** es **diseño+implementación**, una **interfaz** es **sólo diseño**

Utilización de Interfaces (java 7)

- Para que una clase utilice una interfaz debe colocar "***implements***" seguido del nombre de la interfaz, en la declaración de la clase

  ```
  class NombreClase implements NombreInterfaz{ …}
  ```

- **Es posible implementar más de una interfaz en una clase** o heredar de una clase e implementar otra o varias (simula herencia múltiple):

  ```
  class Clase extends CBase implements NomIf1,
    NomIf2 {..}
  ```

  Una clase puede implementar varias interfaces aunque tengan métodos iguales (mismo nombre, número y tipo de parámetros)

- Control de Acceso
  - Todos los métodos de un interfaz son *public* y *abstract* y en las clases que los implementan no se pueden restringir más.
  - Los atributos de una interfaz son *public static final* (constantes)

  Una referencia de una interfaz puede apuntar a cualquier objeto de las clases que la implementan

Ejem. Interfaces

```
interface MiNueva1{
  public void imprimir1();
  void imprimirGrl();
}
```

```
interface MiNueva2{
  int A=33;
  void imprimir2();
  void imprimirGrl();
}
```

```
class Union implements MiNueva1, MiNueva2{
  public void imprimir1(){
    System.out.println ("Uno");
    System.out.println (A);
    //Daría error hacer: A++ porque es cte;
}
  public void imprimir2(){
    System.out.println ("Dos");
  }
  public void imprimirGrl(){}
}
```

```
class PruebaInterface {
  public static void main (String arg[]) {
    Union u=new Union();
    u.imprimir1();
    u.imprimir2();
    u.imprimirGrl();
    MiNueva1 mn= new Union();
  }
}
```

Extensión de interfaces

- Una interfaz se puede extender con extends
- Se puede extender más de una interfaz

```
interface Relucir extends CeraSuelos,
          BrilloBaños {
   double precioSorprendente ();
}
```

- La interfaz *relucir* hereda todos los métodos y constantes de *CeraSuelos* y *BrilloBaños* y los añade a su método *precioSorprendente*

Ejem. Sobrecarga de interfaces (I)

```
class Union implements MiInterfaz{
  public void imprimir1(){
    System.out.println ("Uno: Union");
  }
}
```

```
interface MiInterfaz{
   public void imprimir1();
}
```

```
class UnionSobrecarga implements MiInterfaz {
  public void imprimir1(){
    System.out.println ("Uno: UnionSobrecarga");
  }
}
```

Ejem. Sobrecarga de interfaces (II)

```
class SobrecargaIntf {
  void sobrecargado( MiInterfaz m){
    m.imprimir1();
  }
  public static void main (String arg[]) {
    Union u=new Union();
    UnionSobrecarga us=new UnionSobrecarga();

    SobrecargaIntf intf=new SobrecargaIntf();

    u.imprimir1();
    intf.sobrecargado(u);
    intf.sobrecargado(us);
  }
}
```

Resultado:

```
Uno: Union
Uno: Union
Uno: UnionSobrecarga
```

Ejem. Sobrecarga de interfaces (III)

```
class SobrecargaIntf2 {
  MiInterfaz devuelveInterfaz (int x) {
    MiInterfaz u;
    if (x==0)
      u= new Union();
    else
      u= new UnionSobrecarga();
    return (u);
  }
  public static void main (String arg[]) {
   SobrecargaIntf2 intf= new SobrecargaIntf2();
    Union u2= (Union) intf.devuelveInterfaz(0);
     u2.metodoNoDefinidoEnMiInterfaz();
    u2.imprimir1();
    UnionSobrecarga us2=
       (UnionSobrecarga)intf.devuelveInterfaz(1);
    us2.imprimir1();
  }
}
```

Resultado:

```
Uno: Union
Uno: UnionSobrecarga
```

Ejem. Interfaz comparable

```
public class Alumno implements Comparable<Alumno> {
  String nombre;
  int edad;
  Alumno (String n, int e){
     edad=e;
     nombre=n;
  }
  public String toString () {
     return ( nombre + " " + String.valueOf(edad));
  }
public int compareTo(Alumno a) {  //Ordeno primero por nombre y
   luego por edad
    int compEdades=edad- a.edad;
    int compNombres = nombre.compareTo(a.nombre);
     return compEdades+compNombres *1000;
}
}
```

Ejem. Interfaz comparable II

```
public class VectorAlumnos {
   static void mostrar(Vector  va){
   for (int i=0; i<va.size(); i++)
      System.out.println (va.elementAt(i) );
   }
public static void main (String arg[]){
Vector <Alumno> va;
va=new Vector <Alumno>(2);
Alumno a= new Alumno ("Juan", 11);
   va.add(a);
   va.add(new Alumno ("Ana",12));
   va.add(new Alumno ("Juan",13));
   va.add(new Alumno ("María",14));
   for (int i=0; i<va.size(); i++)
     System.out.println (va.elementAt(i) );

 System.out.println("Ordenado por nombre y después por edad");
   Collections.sort(va);
   VectorAlumnos.mostrar(va);
      //  a=(Alumno) vo.elementAt(1)
}
}
```

Resultado:

Juan 11

Ana 12

Juan 13

María 14

Ordenado por nombre y después por edad

Ana 12

Juan 11

Juan 13

María 14

Interfaces en Java 8

- En java-8 se permiten métodos static en las interfaces
- En java-8 se añadió la posibilidad de codificar métodos por defecto (método default) que se codificarían en la propia interfaz con el objeto de:
  - ***No modificar las clases que usen esa interface***. Si una clase implementa una nueva interfaz no necesita añadir código adicional si basta con implementar el método default en la interfaz
  - ***Simular una "Seudo Herencia Múltiple"***, ya que java no dispone de herencia múltiple como tal, pero si implementamos métodos default en diferentes interfaces podríamos tenerla.

Interfaces con métodos default

```
public class Clase3 implements MiInterfazConDefault{
public void metodo1(){
  System.out.println("Metodo 1");
}
public void metodo3(){
  System.out.println("Metodo 3 en clase 3");
}
public static void main (String arg[]){
 Clase3 c= new Clase3();
 c.metodo1();
 c.metodo2(); //Se ejecuta el de la interfaz
 c.metodo3(); //Se ejecuta el definido en Clase3
 MiInterfazConDefault m= c;
 m.metodo3(); //Se ejecuta el definido en Clase3
}
}
```

```
public interface MiInterfazConDefault {
void metodo1();
default void metodo2(){
  System.out.println("Metodo 2"); }
default void metodo3(){
  System.out.println("Metodo 3");  }
}
```

```
Metodo 1
Metodo 2
Metodo 3 en clase 3
Metodo 3 en clase 3
```

Interfaces en Java 9

- En java-9 se permiten métodos privados (estáticos y no estáticos) no abstractos con el fin de mejorar la reusabilidad. Si 2 métodos por defecto necesitan reutilizar código, puedes poner ese código en un método private de la interfaz que no será visible para las clases que lo implementen.
  - Los métodos privados estáticos pueden ser utilizados en otros métodos de la interfaz(estáticos o no)
  - Los métodos privados no estáticos no pueden utilizarse en métodos estáticos de la interfaz.

Modificadores

- Palabras reservadas que se utilizan para establecer el tipo de acceso a clases, propiedades y métodos.

	Clase	Atributo	Variable	Método
public	Si	Si	No	Si
protected	No	Si	No	Si
private	No	Si	No	Si
static	No	Si	No	Si
abstract	Si	No	No	Si
final	Si	Si	Si	Si

Control de acceso

•Niveles en el control de acceso de miembros (atributo o método) de una clase:

	Clase	Subclase	Paquete	Mundo
private	**X**			
protected	**X**	**X***	**X**	
public	**X**	**X**	**X**	**X**
Por defecto	**X**		**X**	

•Clase= la propia clase tiene acceso al miembro
•Subclase= las clases que heredan, estén o no en mismo paquete tienen acceso al miembro
•Paquete= las clases que están en el mismo paquete,sin importar el parentesco, tienen acceso.
•Mundo= Todas las clases tienen acceso
•X* = salvedad

public, protected, private...

- Un elemento público ("**public**") es accesible directamente, tanto por todas las otras clases de su mismo paquete como por aquellas clases que lo importen.
- Un elemento protegido (“**protected**”) es accesible desde clases del mismo paquete en el que esté o **por métodos del mismo paquete que hereden de la actual ,aunque no estén en el mismo paquete, siempre que la referencia utilizada para el acceso sea de la propia clase extendida**.
- Un elemento privado ("**private**“) sólo puede ser accedido por los elementos de la clase donde esté definido.
- El acceso paquete (el acceso por defecto) permite que un elemento pueda ser accedido desde clase del mismo paquete

Ejem. protected

```
package Griego;

public class Alpha {
  protected int estoyProtegido;
  protected void metodoProtegido() {
    System.out.println(”Met Protegido");
  }
}
```

```
package Griego;
public class Gamma {
   void metodoAccesor() {
     Alpha a = new Alpha();
     a.estoyProtegido = 10; // legal
     a.metodoProtegido(); // legal
   }
}
```

```
package Latin;
import Griego.*;

class Delta extends Alpha {
    void metodoAccesor(Alpha a, Delta d) {
        a.estoyProtegido = 10;    // ilegal
        d.estoyProtegido = 10;    // legal
        a.metodoProtegido();    // ilegal
        d.metodoProtegido();    // legal
    }
}
```

static

- Un elemento estático es un elemento permanente, es decir, que no se crea y. Por ello, se dice que el elemento está asociado a la clase y no a las instancias de esa clase.
- El elemento estático puede ser accedido, tanto a través del nombre de la clase como a través de la referencia a un objeto de esa clase.

Ejem. static

```
class InstanciasNumeros {
  public static void main (String arg[]){
    int i;
    Numero n[]=new Numero[3];
    for(i=0; i<3;i++) {
      n[i]= new Numero();
      System.out.println(n[i].x+" "+n[i].numeroInstancias);
    }
    System.out.println(Numero.numeroInstancias);
    System.out.println(n[0].numeroInstancias);
    System.out.println(Numero.devolverNumeroInstancias());
    System.out.println(n[1].devolverNumeroInstancias());
    System.out.println(n[0].x +" "+ n[1].x +" "+ n[2].x);
}}
class Numero{
  static int numeroInstancias=0;
  int x=numeroInstancias;
  Numero(){ numeroInstancias++;}
  static int devolverNumeroInstancias(){
    return numeroInstancias;
  }
}
```

Salida:

0 1
1 2
2 3
3
3
3
3
0 1 2

Métodos estáticos

- Un método estático **no puede hacer referencia a un objeto concreto** de la clase utilizando el this, **no puede utilizar atributos ni métodos no estáticos utilizando this**
- Un método **estático no puede ser sobreescrito** en una clase extendida **por un** método **no estático**.
- En los métodos estáticos de la clase extendida **no se puede utilizar la palabra super** para hacer referencia al objeto de la clase base ya que es un método estático y no puede hacer referencia a ningún objeto concreto
- El **tipo que se utiliza para determinar** cual de los **métodos** (el de la clase base o el sobreescrito en la extendida) se utiliza **es el tipo de la referencia del objeto.**
- **Los constructores no pueden ser static.**

Ejem. Métodos estáticos

```
class StaticPadre {
  public static String str ="PadreStr";
  public  static void show () {
    System.out.println("ClasePadre.str: "+str);
    }
  }
class StaticHijo extends StaticPadre {
  public static String str ="HijoStr";
  public static void show () {
    System.out.println("ClaseHijo.str: "+str);
// Error: System.out.println (super.str);
// Error: super.show();
  }
public static void main (String[]args) {
  StaticHijo ext = new StaticHijo();
  StaticPadre sup = ext;
  sup.show();
  ext.show();
  System.out.println ("sup.str= "+sup.str);
  System.out.println ("ext.str= "+ext.str);
  }
}
```

Resultado:

```
ClasePadre.str: PadreStr
ClaseHijo.str: HijoStr
sup.str= PadreStr
ext.str= HijoStr
```

Código Estático

```
public class CodigoStatic {
int x=3;
static int sx=33;
static {  //Se utiliza para inicializar atributos estáticos
   System.out.println("Ejecución código estático");
  // No válido usar el atributo no estático : System.out.println(x);
   System.out.println(sx);
}
{ //Se puede usar para inicializar variables no estáticas sino se hace en el constructor
  System.out.println("Algo");  }   //Código no estático ejecutado al crear objetos
}
```

Código Estático II

```
public class ClaseUsaCodigoEstatico {
  public static void main(String[] args) {
     System.out.println("Main");
     CodigoStatic cs= new CodigoStatic();  //Se ejecuta el código estático de CodigoStatic

     CodigoStatic cs2= new CodigoStatic();  //Aquí ya no se ejecuta el código estático de CodigoStatic
  }
}
```

Resultado:

Main

Ejecución código estático

33

Algo

Algo

Métodos y clases finales

- Una **clase final no puede tener subclases**

```
final class NombreClase {. . .}
```

- Si un **método es final**, **ninguna clase extendida puede anularlo** para cambiar su comportamiento. (Si podría haber otro en la misma clase base con distinta signatura (distinto número o tipo de parámetros))
  - Motivo: seguridad (método validarPasssword)
- Un **atributo final no puede modificar el valor que se le asigna** durante su declaración o en el constructor
  - Tiene que definirse el valor de un atributo final o en su declaración o en **todos** los constructores de la clase.
- Todos los **métodos de una clase final son** implícitamente **finales**
- **Los constructores no pueden ser final**
- **Un parámetro, declarado como final no se puede modificar**
  - **void método (final int n) {//daría error hacer n++;}**

Clases finales

```
final class ClaseFinal {
  public String str ="PadreStr";
  public void show () {
    System.out.println("ClasePadre.str: "+str+" "+'\u0043');
  }
  public static void main (String[]args) {
    ClaseFinal obj= new ClaseFinal();
    obj.show();
  }
}

/* Error si declarara
  class ClaseHijo extends ClaseFinal {
  .....
  }
*/
```

Métodos y atributos finales

```
class RectanguloFin{
  float base;
  float altura;
// Opcion A:
//  final float PI=(float)3.1416;

// Opcion B:
  final float PI;
  RectanguloFin (){PI=(float) 3.2;}

  final float area(){
    return base*altura;
  }
}
class CuadradoFin extends RectanguloFin{
  /*En esta clase no puede aparecer un método area con
  la misma signatura (nombre método y número y tipo de
  parámetros) si sólo se diferencian en el tipo
  devuelto también da error*/
}
```

abstract

- Clases abstractas: clases que definen sólo parte de la implementación.
- No pueden ser instanciadas (crear objetos)
- Dejan a las clases extendidas dar la implementación específica a algunos métodos o a todos ellos.
- Una clase con un método abstracto debe ser declarada como abstracta
- Se debe poner `abstract` antes del nombre de clase o método

Ejem. abstract(I)

```
abstract class ClaseAbstr{
  void imprime1(){
    System.out.println("Imprime el 1");
   }
   abstract void imprime2();
}

class ClaseAbstrHija extends ClaseAbstr{
   void imprime2(){
     System.out.println("Imprime el 2");
   }
   public static void main (String a[]) {
     ClaseAbstrHija ca1=new ClaseAbstrHija();
     ca1.imprime1();
     ca1.imprime2();
//No se puede hacer
//     ClaseAbstr ca=new ClaseAbstr();
     ClaseAbstr ca;
     ca=ca1;
     ca.imprime1();
     ca.imprime2();
   }
}
```

```
class AreaPoligonos{
  char seleccionarTipoPoligono(Teclado t)throws IOException{
    char c;
    System.out.print("Dime el tipo de poligono (t/r):");

    c=t.leerChar();
    t.leerFinLinea();
    return c; }
  Poligono seleccionarPoligono(char c){
    Poligono p;
    switch(c){
      case 't': p= new Triangulo(4,3); break;
      case 'r': p= new Rectangulo(4,5);break;
      default: p=null;}
    return p;
  }
  public static void main (String arg[]) throws IOException{
    Poligono p;
    char c;
    Teclado t=new Teclado();
    AreaPoligonos ap= new AreaPoligonos();
    c= ap.seleccionarTipoPoligono(t);
    p= ap.seleccionarPoligono(c);
    if (p!=null)  System.out.println("El area es:"+p.area());
    else System.out.println("No existe ese tipo de
poligono");
  }
}
```

Ejem. Abstract (IIa)

```
abstract class Poligono{
  float base;
  float altura;
  Poligono(float b, float a){
    base=b;
    altura=a;
  }
  abstract float area();
}
```

Ejem. abstract(IIb)

```
class Triangulo extends Poligono{
  Triangulo(float b, float a){
    super(b,a);
  }
  float area(){
    return (base*altura)/2;
  }
}
```

```
class Rectangulo extends Poligono{
  Rectangulo(float b, float a){
    super(b,a);
  }
  float area(){
    return (base*altura);
  }
}
```

Inner class

- Se puede definir una clase dentro de otra clase, si la clase interna no es estática se les denomina inner class.
- La clase interna tiene acceso a todos los atributos y métodos de la clase que la contiene.

Ejemplo inner class

```
public class ClaseExterna {
 int atrExt=99;
 void metExterno(){
        System.out.println("Método clase externa");
        ClaseInterna objInt2= new ClaseInterna(22);
        objInt2.metInterno();
 }
         class ClaseInterna {
         int atrInt=33;
         ClaseInterna(int x){
                  atrInt=x;
        }
         void metInterno(){
                  System.out.println("Método clase
                    interna");
                  System.out.println("Atr externo:"+ atrExt);
         }
         void metInterno2(){
                 System.out.println("Atr interno:"+ atrInt);
                 metExterno();
          }
        }//fin claseInterna
```

```
public static void main (String arg[]){
  ClaseExterna objExt= new ClaseExterna();
  objExt.metExterno();
 //No es válido usar:
//ClaseInterna objInt= new ClaseInterna(33);
  ClaseInterna objInt= objExt.new ClaseInterna(11);
  objInt.metInterno2();
  }
} //fin claseExterna
```

Resultado ejecución:

Método clase externa
Método clase interna
Atr externo:99
Atr interno:11
Método clase externa
Método clase interna
Atr externo:99

Inner class en métodos

- También se pueden definir inner class en métodos.
- Dentro de la inner class se tiene acceso a las variables del método que la contiene y a los atributos.
- No se pueden definir miembros static, que no sean final, dentro de la inner class

Ejemplo inner class en método

```
public class ClaseInternaEnMetodo {
 int atrExt=11;
 void metExterno2(){
        int var1=22;
        final int var2=33;
        System.out.println("Método 2 clase externa");
        //No puedo crear objetos de la clase interna hasta
              después de definirla
        // Error: ClaseInterna2  ob=new ClaseInterna2();
      class ClaseInterna2{
       int atrInt=33;
            void metInterno(){
                  System.out.println("Método clase interna");
                  System.out.println("Atr externo:"+ atrExt);
                  System.out.println ("var1 externa:"+ var1);
                  System.out.println("var2 externa:"+var2);
             }
        }
     System.out.println("Creo la clase interna");
     ClaseInterna2  o2=new ClaseInterna2();
     o2.metInterno();
     System.out.println("o2.atrInt:"+o2.atrInt);
 }//fin metExterno2
 void metExterno1(){
System.out.println("Método 1 clase externa");
//Aquí ya no es accesible la clase interna:
//Error: ClaseInterna2  oc=new ClaseInterna2();
  }
 public static void main (String arg[]){
  ClaseInternaEnMetodo objExt= new
ClaseInternaEnMetodo();
  objExt.metExterno1();
  objExt.metExterno2();
  }
} //fin ClaseInternaEnMetodo
```

Resultado ejecución:

```
Método 1 clase externa
Método 2 clase externa
Creo la clase interna
Método clase interna
Atr externo:11
Var1 externa:22
var2 externa:33
o2.atrInt:33
```

Anonymous Inner class

- En gestión de eventos se suelen utilizar inner class sin nombre: **anonymous inner class**.
- Las clases anónimas pueden extender de una clase o implementar una interface.
- La sintáxis es:

```
new nombreClaseHeredadaoInterfazImplementado
    ([parámetros para el constructor]) {
    //  instrucciones de la clase
}
```

Ejemplo anonymous inner class

```
public class UsoClaseInternaAnonima {
  public static void main(String[] args) {
    ProbarVideo pv= new ProbarVideo();
    pv.prueba (new Video2());
     pv.prueba(
       new Video (){
         public void play(){System.out.println("Reproducción");};
         public void stop(){System.out.println("Parada");};
       } //fin  Video
    ); // Fin pv.prueba
}
}
```

```
public interface Video {
  void play();
  void stop();
}
```

```
public class Video2 implements Video {
  void play() {
System.out.println("Reproducción2");}
  void stop() {
System.out.println(“Parada2");}
}
```

```
public class ProbarVideo {
  void prueba (Video v){
      v.play();
      v.play();
      v.stop();
  }
}
```

Resultado ejecución:

Reproducción2
Reproducción2
Parada2
Reproducción
Reproducción
Parada

Ejem. Anonymous inner class

```
import javax.swing.*;
import java.awt.event.*;

public class ActionListenerTest3 {
 public static void main(String[] args) {
  JFrame.setDefaultLookAndFeelDecorated(true);
  JFrame frame = new JFrame();
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  JButton button = new JButton("Selecciona el fichero");
  button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
     JFileChooser fileChooser = new JFileChooser();
     int returnVal = fileChooser.showOpenDialog(null);
     if (returnVal == JFileChooser.APPROVE_OPTION) {
       System.out.println(fileChooser.getSelectedFile().getName());
     }
    }
 });
frame.add(button);
frame.pack();
frame.setVisible(true);
  }//main
}
```