Unidad 4. POO. Clases.

Contenido

Objetivos:

Emiliano Torres

- Diseñar e implementar la estructura y miembros de una clase.
- Trabajar en profundidad con el concepto de clase.
- Aplicar el concepto de **herencia** en la resolución de problemas.
- Comprender el concepto de **recursividad** y saber aplicarlo en la resolución de ejercicios.
- Agrupar los programas y clases generadas en paquetes para crear una estructura más lógica y útil.

Bibliografía:

- ***Programación***. Juan Carlos Moreno Pérez. RA-MA, 2011 ISBN 978-84-9964-088-4

1. Relación de herencia

Mediante la palabra clave **extends** podemos heredar las características de una clase ya existente.

Sintaxis de la herencia:

```
modificador_acceso class NombreSubclase extends NombreSuperclase
{
}
```

De esta manera nos podemos ahorrar escribir código cuando ya existe un objeto similar.

Al desarrollar una clase ("hijo" o subclase) que desciende de otra ("padre" o superclase), dicha clase hereda los campos y métodos de la clase padre. También podemos sobrescribir métodos de la clase padre para cubrir mejor las necesidades de la clase hijo, así como ocultar campos, etc. A la hora de sobrescribir los métodos de la clase padre añadiremos la anotación **@Override**, lo cual hace que el compilador compruebe que estamos sobrescribiendo un método existente en la clase padre.

Se basa en la existencia de relaciones de **generalización/especialización** entre clases.

Las clases se disponen en una jerarquía, donde una clase hereda los atributos y métodos de las clases superiores en la jerarquía.

Una clase puede tener sus propios atributos y métodos adicionales a lo heredado.

Una clase puede modificar los atributos y métodos heredados.

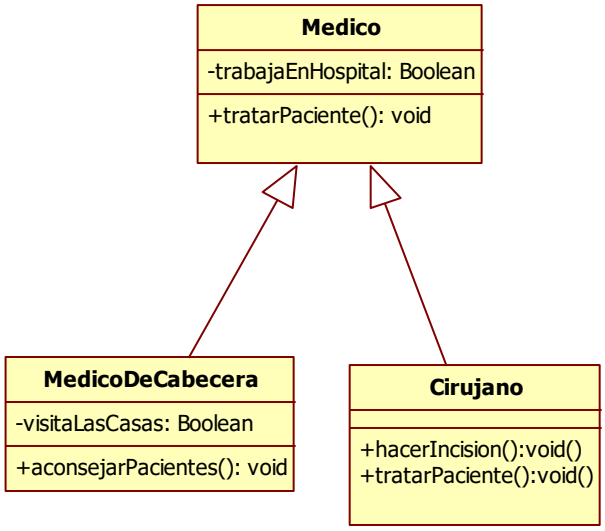
Las clases por encima en la jerarquía a una clase dada, se denominan **superclases**.

Las clases por debajo en la jerarquía a una clase dada, se denominan **subclases**.

Una clase puede ser superclase y subclase al mismo tiempo.

Tipos de herencia:

- Simple
- Múltiple (no soportada en Java)




```
public class Medico
{
  private boolean trabajaEnHospital;

  public void tratarPaciente()
  {
      //Realizar un chequeo
  }
}

public class MedicoDeCabecera extends Medico
{
  private boolean visitaLasCasas;

  public void aconsejarPaciente()
  {
      //Ofrecer remedios caseros
  }
}

public class Cirujano extends Medico
{
  public void tratarPaciente()
```

```
  {
      //Realizar una operación
  }
  public void hacerIncision()
  {
      //Realizar incisión
  }

}
```

Conteste las siguientes preguntas basándose en el ejemplo anterior:

¿Cuántos atributos tiene la clase Cirujano?

¿Cuántos atributos tiene la clase MedicoCabecera?

¿Cuántos métodos tiene la clase Medico?

¿Cuántos métodos tiene la clase Cirujano?

¿Cuántos métodos tiene la clase MedicoDeCabecera?

¿Puede un médico MedicoDeCabecera tratar pacientes?

¿Puede un médico MedicoDeCabecera hacer incisiones?

Soluciones: 1-2-1-3-2-Si-No

Tipos de clase

En el ejemplo anterior (public class MiClase) el modificador era public. Estos modificadores de clase pueden ser:

- **public** Son accesibles desde otras clases, ya sea de manera directa o por herencia. Para que puedan acceder a las clases que están en paquetes externos antes hay que importar estos.
- **abstract** Estas clases poseen, al menos, un método abstracto. Estas clases abstractas no se instancian, sino que se utilizan como base para la herencia.
- **final** Es la clase que termina una cadena de herencia. Por tanto, es lo contrario de las clases abstractas. No pueden heredarse.
- **synchronizable** Indica que todos los métodos de la clase son sincronizados, lo que significa que no puede accederse a ellos al mismo tiempo desde distintas tareas. Gracias a esto podemos modificar las mismas variables desde distintas tareas sin temor a que se sobrescriban.
- **sin modificador** La clase puede ser usada e instanciada por clases dentro del package donde se define.

Modificadores de acceso

Los modificadores de acceso permiten al diseñador de una clase determinar quién accede a los datos y métodos miembros de una clase.

Los modificadores de acceso preceden a la declaración de un elemento de la clase (ya sea dato o método), de la siguiente forma:

[modificadores] tipo_variable nombre;

[modificadores] tipo_devuelto nombre_Metodo (lista_Argumentos);

Existen los siguientes modificadores de acceso:

- **public** - Todo el mundo puede acceder al elemento. Si es un dato miembro, todo el mundo puede ver el elemento, es decir, usarlo y asignarlo. Si es un método todo el mundo puede invocarlo.
- **protected** – Se puede acceder desde la propia clase y subclases. También desde cualquier clase del package donde se define la clase.
- **sin modificador** (o package) - Se puede acceder al elemento desde cualquier clase del package donde se define la clase.
- **private** - Sólo se puede acceder al elemento desde métodos de la clase, o sólo puede invocarse el método desde otro método de la clase.

En el ejemplo anterior se definía una variable miembro en MiClase a través de “int i;”, en este caso no se ha definido modificador

	Class	**Package**	**Subclass**	**World**
public	Y	Y	Y	Y
protected	Y	Y	Y	N
sin modificador	Y	Y	N	N
private	Y	N	N	N

Visibilidad entre los elementos de la clase:

	private	**sin modificador**	**protected**	**public**
Desde la misma clase	Y	Y	Y	Y
Desde una subclase de paquete	N	Y	Y	Y
Desde una no-subclase de paquete	N	Y	Y-->N	Y
Desde una subclase de diferente paquete	N	N	Y	Y
Desde una no-subclase de diferente paquete	N	N	Y	Y

Guía para usar el control de acceso:

- Usar private para métodos y variables que solamente se utilicen dentro de la clase y que deberían estar ocultas para todo el resto.
- Usar public para métodos, constantes y otras variables importantes que deban ser visibles para todo el mundo.
- Usar protected si se quiere que las clases del mismo paquete y subclases puedan tener acceso a estas variables o métodos.
- No usar nada, dejar la visibilidad por defecto (default, package) para métodos y variables que deban estar ocultas fuera del paquete, pero que deban estar disponibles al acceso desde dentro del mismo paquete. Utilizar protected en su lugar si se quiere que esos componentes sean visibles fuera del paquete.

[Problemas resueltos de programación en lenguaje java paso a paso]

Cuando una clase hereda de otra, la clase hija hereda los atributos y los métodos de la clase padre, pero existen ciertas restricciones en esta herencia:

- Los atributos y los métodos con modo de acceso `private` no se heredan.
- Se heredan los atributos y métodos con modo de acceso `public y protected`.
- Se heredan los atributos y métodos con modo de acceso `package` si la clase padre y la clase hija pertenecen al mismo paquete. `package` es el modo de acceso por defecto, por lo que no hay que especificar la palabra reservada `package`.
- No se hereda un atributo de una clase padre si en la clase hija se define un atributo con el mismo nombre que el atributo de la clase padre.
- No se hereda un método si éste es sobrecargado.

Ejemplo de uso:

Crea un proyecto y un dentro de este package crearemos una clase llamada PruebaModificadoresAcceso y copiaremos el siguiente código:

```
class Profesor
{
	String nombre;
	int edad;
	protected int añosAntiguedad;
}
class Catedratico extends Profesor //ESTO CREA UNA SUBCLASE
{
	int añosCatedratico;
	public double obtenerSalario()
	{
		return (925 + añosAntiguedad * 33.25 + 47.80 * añosCatedratico);
	}
	public void imprimirSalario()
	{
	System.out.println("El salario de "+nombre+" de "+edad+" años es "+obtenerSalario()+" €");
}
}
public class PruebaModificadoresAcceso {
	public static void main(String[] args) {
		Catedratico catedratico=new Catedratico();
		catedratico.nombre="Paco";
		catedratico.edad=47;
		catedratico.añosAntiguedad=10;
```

```
            catedratico.añosCatedratico=5;
            catedratico.imprimirSalario();
        }
}
```

Ahora prueba lo siguiente:

1- Cambia String nombre; de la linea 6 a **private** String nombre;

   Observa que Eclipse nos indica errores de que el campo ya no puede ser utilizado en subclases porque no es visible.

2- Cambia ahora **private** String nombre; de la linea 6 a **protected** String nombre; Observa que ahora Eclipse no da error, ya que ese campo ahora es accesible por todas las sublcases (Catedratico en nuestro caso) y las clases del package

El uso de la Herencia

Debemos usar herencia cuando hay una clase de un tipo más específico que una superclase. Es decir, se trata de una especialización.

- Lobo es más específico que Canino. Luego tiene sentido que Lobo herede de Canino.

Debemos usar herencia cuando tengamos un comportamiento que se puede reutilizar entre otras clases del mismo tipo genérico.

- Las clases Cuadrado, Círculo y Triángulo tienen que calcular su área y perímetro luego tiene sentido poner esa funcionalidad en una clase genérica como Figura.

No debemos usar herencia sólo por el hecho de reutilizar código. Nunca debemos romper las dos primeras reglas.

- Podemos tener el comportamiento cerrar en Puerta. Pero, aunque necesitemos ese mismo comportamiento en Coche no vamos a hacer que Coche herede de Puerta. En todo caso, coche tendrá un atributo de tipo Puerta.
- No debemos de usar herencia cuando no se cumpla la regla: Es-un(Is-a)

¿Refresco es una Bebida? Si, la herencia tiene sentido.

¿Bebida es un Refresco? No, la herencia no tiene sentido.

La sintaxis de Java para declarar jerarquías es:

```
class Vehiculo {
// Definicion de métodos y atributos que posean todos los vehículos
}
class VehículoConMotor extends Vehiculo {
        // Cualquier componente no listada se ``hereda '' de Vehiculo
        // excepto el constructor
        // Atributos declarados en esta clase
        // Constructor(es)
        //Metodos de la clase Vehiculo redefinidos
        //Metodos adicionales que son exclusivos de VehículoConMotor
        //Metodos privados adicionales
```

```
}
```

Ejemplo de uso para Herencia de clases

```
package tema4;
//Parent class that represents a basic
//vehicle feature and function.
class Vehiculo
{
	private int numRuedas = 4;
	private int numPuertas=4;
	protected boolean llevoRadio=true;
	public Vehiculo()
	{
		showinfo();
	}
public Vehiculo(int numPuertas,int numRuedas)
{
	this.numPuertas=numPuertas;
	this.numRuedas=numRuedas;

	showinfo();
}
public void showinfo()
{
	System.out.println("Soy un vehiculo de "+ numPuertas +" puertas y "+numRuedas+" ruedas");
}
public void conducir()
{
	System.out.println("Conduzco con mis " + numRuedas + " ruedas.");
	if (llevoRadio) System.out.println("Voy escuchando la radio");
}
}
//Subclase que añade un método y sobreescribe uno existente
class Coche extends Vehiculo
{
	public Coche()
	{
		System.out.println("Soy un coche");
```

```
	}
	@Override
	public void conducir()
	{
		cierraPuertas();
		super.conducir();
	}
	public void cierraPuertas()
	{
		System.out.println("Cierro las puertas.");
	}
}
//Subclase que sobreescribe el valor de un campo y hace una llamada a un
constructor especifico para inicializar otras variables
class Moto extends Vehiculo
{
	public Moto() {
		super(0,2);
		//super();
		llevoRadio=false;
		System.out.println("Soy una moto");
	}
	public Moto(int x , int y) {
		super(x,y);
		//super();
		llevoRadio=false;
		System.out.println("Soy una moto");
	}

}
//Clase principal para probarlo
public class VehiculoTest
{
	public static void main(String[] args)
	{
		Coche _coche = new Coche();
		_coche.conducir();
		Moto _moto = new Moto();
```

```
                _moto.conducir();
        }
}
```

La salida es:

```
Soy un vehículo de 4 puertas y 4 ruedas
Soy un coche
Cierro las puertas.
Conduzco con mis 4 ruedas.
Voy escuchando la radio

Soy un vehículo de 0 puertas y 2 ruedas
Soy una moto
Conduzco con mis 2 ruedas.
```

Como puede verse en este ejemplo, podemos:

- cambiar los valores con los que se inicializa la clase padre llamando al constructor adecuado.
- Cambiar los valores de campos del padre directamente sobrescribiéndolos desde la clase hijo
- Cuando hemos sobrescrito un método del padre, y necesitamos llamar a la versión del padre de este método, usamos la palabra clave **super** y el operador punto.
- Para evitar la confusión entre la variable de método y la variable miembro usamos la palabra clave **this**

NOTA:

En el ejemplo hemos creado todas las clases (Vehiculo, Coche, Moto y VehiculoTest) dentro del mismo fichero VehiculoTest.java

Esto lo hacemos por comodidad para aprender los objetivos de cada apartadao, aunque en el mundo profesional las clases suelen ir separadas cada una en su fichero y package correspondiente.

Nota:

Acostumbramos a preceder las variables de una clase con underscore, x ejemplo protected String _cadenaCaracteres; y así diferenciar los atributos de una clase con los que se reciben como parámetros, por ejemplo.

2. super

super es una referencia del objeto actual, pero apuntando al padre.

super se utiliza para acceder desde un objeto hijo a atributos y métodos (incluyendo constructores) del padre.

Cuando el atributo o método al que accedemos no ha sido sobreescrito en la subclase, el uso de super es redundante.

Los constructores de las subclases incluyen una llamada a super() si no existe un super o un this.

Ejemplo de **acceso a un atributo**:

```
public class ClasePadre
{
```

```
   public boolean atributo = true;
    public void imprimir()  {
System.out.println(atributo);
}
}
public class ClaseHija extends ClasePadre{
    private boolean atributo = false;
@Override
    public void imprimir()
    {
        super.imprimir();
        System.out.println(super.atributo);
    }
}
public class TestSuper
{

    public static void main (String[] args)
    {
        ClaseHija ch = new ClaseHija();
        ch.imprimir();

    }
}
```

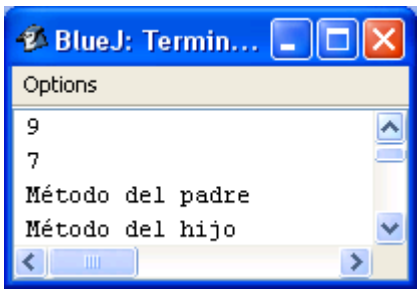
Ejemplo de **acceso a un constructor**:

```
public class ClasePadre {
   …
   public ClasePadre(int parametro) {
       System.out.println(parametro);
   }
}
public class ClaseHija extends ClasePadre{
      …
    public ClaseHija(int parametro) {
        super(parametro+2); //tiene que ser la primera línea del constructor
y sólo puede usarse una vez por contructor
        System.out.println(parametro);
    }
}
public class TestSuper
{
    public static void main (String[] args)
    {
        ClaseHija ch = new ClaseHija(7);
    }
}
```

Ejemplo de **acceso a un método**:

```
public class ClasePadre {
    …
    public void imprimir() {
        System.out.println("Método del padre);
    }
}
public class ClaseHija extends ClasePadre{
       …
     public void imprimir()    {
         super.imprimir();
         System.out.println("Método del hijo");
     }
}
public class TestSuper
{

     public static void main (String[] args)
     {
         ClaseHija ch = new ClaseHija(7);
         ch.imprimir();
     }
}
```



3. this

this es una referencia al objeto actual.

this se utiliza para acceder desde un objeto a atributos y métodos (incluyendo constructores) del propio objeto.

Existen dos ocasiones en las que su uso no es redundante:

- Acceso a un constructor desde otro constructor. Ejemplo:

```
public class MiClase{
    public MiClase() {
        this(2); //MiClase(2)
        System.out.println("Constructor sin");
   }
public MiClase(int parametro)
        System.out.println("Constructor con " + parametro);
   }
}
Salida:
      Constructor con 2
      Constructor sin
Otro ejmplo:
public class MiClase
{   private int dato;
    public MiClase() {
        this(0); //dato = 0;
   }
    public MiClase(int parametro)
        dato = parametro;
         System.out.println("Atributo inicializado a "+ dato);
   }
}
MiClase mc1 = new MiClase();
System.out.println(mc1.getDato());
MiClase mc2 = new MiClase(2);
System.out.println(mc2.getDato());
```

- Acceso a un atributo desde un método donde hay definida una variable local con el mismo nombre que el atributo. Ejemplo:

```
public class MiClase
{
    private int x = 5;
    public void setX(int x)
    {
        System.out.println("x local vale: " + x);
        System.out.println("x atributo vale: " + this.x);
        this.x = x;
        System.out.println("x atributo vale: " + this.x);
   }
}
MiClase mc = new MiClase();
mc.setX(3);
Salida:
      x local vale: 3
      x atributo vale: 5
      x atributo vale: 3
```

- Devolver referencias al propio objeto.

Ejemplo:

```
public class Rectangulo extends Figura
{
    private int ancho, alto;

    public Rectangulo(int ancho, int alto)
    {
        this.ancho = ancho;
        this.alto = alto;
   }
    public int getAncho()
    {
        return this.ancho; //se puede omitir this
    }
    public int getAlto()
    {
        return alto;
    }
```

```
    public Rectangulo incrementarAncho()
    {
        ancho++;
        return this;
    }
    public Rectangulo incrementarAlto()
    {
        this.alto++; //se puede omitir this
        return this;
    }
}
```

4. La clase Object

La clase Object es la raíz de la jerarquía de clases en Java.

Es la clase raíz del árbol de clases Java. Todas las clases heredan de esta. Proporciona una serie de métodos básicos que heredan todas las clases descendientes. Algunos de ellos son:

En Java todas las clases heredan de otra clase:

- Si se especifica en el código con la palabra reservada `extends`, la clase heredará de la clase especificada.
- Si no lo especificamos en el código, el compilador hace que la clase herede de la clase Object.

Ejemplo:

```
public class MiClase extends Object
{
  //No hace falta pues el compilador lo hace automáticamente
}
```

Esto significa que las clases siempre van a contar con los atributos y métodos de la clase Object.

Algunos de sus métodos más importantes son:

- **`public boolean equals(Object o);`**

  ```
  Object miObjeto = new Object();
  Coche c = new Coche();
  miObjeto=c;
  //if(c.equals(miObjeto))
  if(c==miObjeto)
  SOP("son iguales");
  else SOP("no son iguales");
  ```

  La implementación de equals en la clase Object sólo comprueba si el receptor y el argumento hacen referencia al mismo objeto:

  La implementación de este método en la clase Object comprueba igualdad de referencias, **no de contenido.**

  Para poder comparar bien nuestros objetos hace falta incluir en nuestra clase la redefinición correcta de boolean equals(Object o).

Permite comparar dos objetos, pero no de la misma manera que el operador ==, que solo compara si las dos referencias apuntan al mismo objeto. Equals devuelve true si ambos objetos son del mismo tipo y contienen los mismos datos, y false si no es así.

Se limita a comprobar que ambas **referencias** son iguales, no hace una comprobación exhaustiva.

Hace lo mismo que ==.

```
Rectangulo r1 = new Rectangulo(3, 2);
Rectangulo r2 = new Rectangulo(3, 2);
Rectangulo r3;
r3=r1;
if(r1.equals(r2) || r1 == r2)
    System.out.println("Iguales r1 y r2");
 else
    System.out.println("No son iguales r1 y r2");

if(r1.equals(r3) || r1 == r3)
    System.out.println("Iguales r1 y r3");
 else
    System.out.println("No son iguales r1 y r3");
```

Ejemplo:

```
class Circulo {
      private int xcentro;
      private int ycentro;
      private int radio;

      public Circulo ( ) {}
      public void ponerCentro ( int x , int y ) {}
      public void ponerRadio ( int r ) {}
      public double longitud( ) { return 0;}
      public double area( ){return 0;}
      public void dibujar( ) {}
      public boolean equals ( Object o ) {
            boolean resultado;
            // Hay que hacer una conversión explícita ( downcasting )
            Circulo c = (Circulo) o;
            if ( this.xcentro != c.xcentro)
                  resultado = false;
            else
                  if ( this.ycentro != c.ycentro )
```

```
                        resultado = false;
                  else
                        if (this.radio != c.radio)
                              resultado = false;
                        else resultado = true;

            return resultado;
      }
}
```

Ejemplo:

```
class Complejo {
private double real ;
private double imag ;

public Complejo(double re, double im ) {
      real = re;
      imag = im;
}
public boolean equals ( Object o ) {
      // Hay que hacer unaconversión explícita ( downcasting )
      Complejo n = (Complejo) o ;
      boolean iguales = false ;
      if ( n.real == this.real  &&  n.imag == this.imag )
            iguales = true ;
      return iguales;
}

//Programa p r i n c i p a l
public static void main ( String [ ] args )
{
      Complejo c1 , c2;
      c1 = new Complejo (2.4,1.2);
      c2 = new Complejo (2.5,3.2);
      if ( c1.equals ( c2 ) )
            System.out.println( "Son iguales " ) ;
      else
            System.out.println( "No son iguales " ) ;
}
}
```

- `public String getClass();`

  Devuelve la clase de la cual es instancia el objeto.

  Podemos utilizar este método para obtener la clase a la que pertenece un objeto.

  Devuelve un objeto de tipo Class con información importante del objeto que crea la clase.

  Este método no puede ser sobrescrito.

  Ejemplo:

```
System.out.println(r1.getClass());
System.out.println(r2.getClass());
System.out.println(r3.getClass());
```

  Salida:

```
class Rectangulo
class Rectangulo
class Rectangulo
```

```
r1.getClass() == r2.getClass() -> true
```

- `public int hashCode();`

  Devuelve un identificador unívoco después de aplicarle un algoritmo **hash**.

  Tarea: buscar información sobre algoritmos hash.

  Ejemplo:

```
System.out.println(r1.hashCode());
System.out.println(r2.hashCode());
System.out.println(r3.hashCode());
```

  Salida:

```
1023040
14372770
1023040
```

  https://es.wikipedia.org/wiki/HashCode()_(Java)

//comprobar que sale lo mismo aplicando el algoritmo hash

- `public String toString();`

  Convierte el objeto en cuestión en una cadena.

  Devuelve la representación visual de un objeto en forma de cadena: nombre de la clase, '@' y la representación hexadecimal del código **hash** del objeto.

  Ejemplo:

```
Rectangulo r1 = new Rectangulo(3, 2);
Rectangulo r2 = new Rectangulo(3, 2);
Rectangulo r3;
r3=r1;
System.out.println(r1.toString());
```

```
System.out.println(r2.toString());
System.out.println(r3.toString());
```

Salida:

```
Rectangulo@64ab4d
Rectangulo@12a55aa
Rectangulo@64ab4d
```

devuelve una cadena con el tipo del objeto y un valor entero del código hash (función sobre los atributos del objeto que devuelve un entero) en hexadecimal.

Para poder obtener una representación en forma de cadena de nuestros objetos hace falta incluir en nuestra clase la redefinición correcta de toString.

```
class ProgramaLavado {
	private String nombre;
	private int numfases;
	private int duracion;
	private int consumo_agua;
	private boolean centrifugado;
	private boolean prelavado;

	public ProgramaLavado ( /*args*/) { }
	public void activar ( ) {  }
	public int consumoAgua ( ) {  }
	public String nombre ( ) {  }
	public int duracion ( ) {  }

	public String toString ( ) {
		String cadena ;
		cadena = "Programa : " + nombre + " , " ;
		cadena += "Duración : " + duracion + " , " ;
		cadena += "Consumo agua : " + consumoAgua() ;
		return ( cadena ) ;
	}
}
```

La redefinición del método toString() en cada clase que definamos substituye a los métodos mostrar().

```
class Complejo {
	private double real ;
	private double imag ;

	public Complejo(double re, double im ) {
		real = re;
		imag = im;
```

```
        }
        public String toString( ) {
              String cadena ;
              cadena = real + " " + imag + " i ";
              return ( cadena ) ;
        }
        //Programa p r i n c i p a l
        public static void main ( String [ ] args )
        {
              Complejo c1 , c2 , c3 ;
              c1 = new Complejo (2.4,1.2);
              System.out.println( c1 ) ;
        }
}
```

- `public Object clone();`

  Devuelve una copia del objeto en otro.

  Utilizar este método equivale a utilizar un **constructor de copia**.

  En muchos casos es necesario implementar un método clone, el cual sobreescriba al método clone de su superclase y actúe de una forma más específica que el método genérico clone().

  Ejemplo:

```
public class Rectangulo implements Cloneable
{
    private int ancho, alto;
    private String nombre;

    public Object clone() {
        Object objeto = null;
        //try{
            objeto = super.clone();


        /*}catch(CloneNotSupportedException ex){
            System.out.println("Error al duplicar");
        }*/
        return objeto;
    }
…  //lo demás igual
}
```

```
public class testFiguras
{

 public static void main (String[] args)
 {
 Cuadrado c = new Cuadrado (5);
 Rectangulo r1 = new Rectangulo(3, 2);
 Rectangulo r2 = new Rectangulo(3, 2);
 Rectangulo r3;

 Rectangulo r4 = (Rectangulo) r1.clone();

 System.out.println("Ancho r4 antes = " + r4.getAncho());
 System.out.println("Alto r4 antes = " + r4.getAlto());

 r4.incrementarAncho();

 System.out.println("Ancho r1 = " + r1.getAncho());
 System.out.println("Alto r1 = " + r1.getAlto());

 System.out.println("Ancho r4 = " + r4.getAncho());
 System.out.println("Alto r4 = " + r4.getAlto());
```

La clase objeto de la clonación deberá de implementar la interfaz cloneable. Si no se implementa esta interfaz, el programa lanzará un excepción de tipo CloneSupportedException. También se ha implementado el método clone(), el cual hace una llamada al método clone() de su clase base.

Muchas veces es más cómodo para el programador utilizar el constructor de copia que el método clone().

Referencias sobre **`implements`**:

http://es.wikipedia.org/wiki/Interfaz_%28Java%29

Anexo A. Implements

Otros métodos:

- `public void finalize();`

  Método al que llama el Garbage Collector para liberar memoria ocupada.

  Si el programador necesita realizar una acción una vez destruido un objeto deberá reescribir este método.

- ```
  public void wait();
  public void notify();
  public void notifyAll();
  ```

  Tienen que ver con el manejo de **threads**.

Otro ejemplo:
  ```

```
public class MiClase
{
}
public class TestMiClase
{
  pblic static void main(String[] args)
  {
    MiClase mc = new MiClase();
    System.out.println("mc: "+ mc);
    System.out.println("toString " + mc.toString());
    System.out.println("hashCode " + mc.hashCode());
    System.out.println("getClass " + mc.getClass());
    System.out.println("equals():" + mc.equals(mc));
  }
}
```

Java al sobrescribir métodos no nos permite reducir la visibilidad, por ello nos impone las siguientes reglas:

- Métodos declarados public en la superclase también deben ser declarados public al sobrescribirlos en las subclases
- Métodos declarados protected en la superclase deben ser declarados public o protected al sobrescribirlos en las subclases. Nunca pueden ser private
- Métodos declarados sin modificador pueden ser declarados public, protected o sin modificador en las subclases.
- Métodos declarados private no pueden ser sobrescritos nunca, ni tan siquiera son heredados.

5. Encapsulación y visibilidad.

5.1. Interfaces.

Las interfaces son clases abstractas puras (no tienen implementación para ninguno de sus métodos).

Los métodos componen la interfaz del objeto con el mundo exterior.

Una interfaz es un grupo de métodos con sus cuerpos vacíos.

Los interfaces son un tipo de clase especial que no implementa **ninguno** de sus métodos. Todos son abstractos. Por tanto, no se pueden instanciar.

Sintaxis de la declaración de un interfaz:

```
modificador_acceso interface nombre_interfaz [extends Interface1, …, InterfaceN] {
        tipo1 metodo1(… args …);//no se implementan los métodos
        tipo2 metodo2(… args …);
        …
}
```

Todos los métodos declarados en una interfaz son public y abstract, sin ser necesario especificarlo. Esto se debe a que son accedidos desde fuera y no se aporta implementación.

En la declaración de una interfaz sólo se pueden incluir declaraciones de atributos constantes que son definidos por defecto como **public, static**(no asociados a instancias) **y final**(son constantes).

Dentro de la interfaz no se pueden utilizar algunos modificadores como transient, volatile, synchronized, private y protected.

Ejemplo:

```
public interface InterfazFigura {
      int area();
}
```

Ejemplo:

```
public interface MiInterfaz {
        public final double PI = 3.14;
        public final int CONST = 75;
        void put(int dato);
        int get();
}
```

Como se puede observar sólo se deben declarar los métodos, pero no se realiza su implementación. Una clase que utiliza una interfaz debe implementar todos los métodos de esa interfaz.

De los interfaces también se hereda, aunque se suele decir implementa. Y se realiza mediante la palabra reservada ***implements***.

Sintaxis de la declaración de herencia:

```
modificador_acceso class nom_clase implements nombre_interfaz {
      public tipo1 metodo1(… args …){
            //implementación
      }
      …
}
```

Para implementar las interfaces de la clase usaremos **implements**, y las distintas interfaces separadas por comas. Una interface puede heredar de otros interfaces.

La interfaz define el método área, para su posterior desarrollo en las clases que implementen esta interfaz, por ejemplo, la clase Rectángulo.

```
public class Rectangulo extends Figura implements Cloneable, InterfazFigura {
      …
      public int area(){
            return ancho * alto;
      }
}
```

Para compilar correctamente una clase que implementa una interfaz, ésta debe contener **todos** los métodos declarados en dicha interfaz.

Una interfaz se puede utilizar como un tipo de datos en métodos y declaraciones, sin embargo, como no se pueden utilizar objetos de una interfaz directamente, se han de utilizar objetos de clases que implementen la interfaz.

Las interfaces siguen siendo clases Java por lo que su código fuente se guarda en un fichero texto de extensión .java y al compilarlo se genera un .class.

5.2. Clases abstractas

Una clase abstracta declara la existencia de métodos pero no la implementación de todos sus métodos (o sea, las llaves { } y las sentencias entre ellas).

Una clase abstracta puede contener métodos no-abstractos pero al menos uno de los métodos debe ser declarado abstracto.

Para declarar una clase o un método como abstractos, se utiliza la palabra reservada **abstract**.

Declaración de un método abstracto:

```
modificador_acceso abstract tipo_retorno nombre([tipo param,…]);
```

Ejemplo:

```
public abstract void miMetodo();
```

Ejemplo:

```
Public abstract class Drawing
{
        public abstract void miMetodo(int var1, int var2);

        public String miOtroMetodo( ) {
                return null;
        }
}
```

Una clase abstracta no se puede instanciar, pero si se puede heredar y las clases hijas serán las encargadas de agregar la funcionalidad a los métodos abstractos. Si no lo hacen así, las clases hijas deben ser también abstractas.

La idea de las interfaces es bastante parecida a la de clase abstracta, ya que no se pueden crear objetos y no existe implementación de métodos.

El objetivo de un método abstracto es forzar una interfaz (API) pero no una implementación.

Una clase puede heredar de múltiples interfaces por lo que simula la herencia múltiple.

Una clase puede heredar de otra clase y a la vez heredar de múltiples interfaces.

Una interfaz puede también definir constantes.

Si una clase que hereda de una interfaz, no implementa todos los métodos de este, deberá ser definida como **abstracta**.

Ejemplo:

```
public abstract class Canino{
        private int años;
        public abstract void comer();
        public abstract void hacerRuido();
```

```
        public int getAños(){ return años;}
    }

    public interface Mascota {
        public void jugar();
        public void vacunar();
        public static final int IVA = 4;
    }
    public class Perro extends Canino implements Mascota {
        private String nombre;

        public Perro(){    }
        public Perro(String nombre, int años){    }

        public void comer() {    }
        public void hacerRuido() {    }

        public void jugar() {    }
        public void vacunar() {    }

        public void rugir() {    }
    }
```

Una interface se trata como un tipo cualquiera.

Por tanto, cuando se habla de polimorfismo, significa que una instancia de una clase puede ser referenciada por un tipo interface siempre y cuando esa clase o una de sus superclases implemente dicho interface.

Ejemplo de uso:

```
interface IProfesor {
        public void identificarse();
}
abstract class ProfesorBase implements IProfesor {
        protected String nombre;
        public void setNombre(String nombre)
        {
                this.nombre= nombre;
        }
        public void identificarse() {
                System.out.println("Me llamo "+this.nombre);
                masInfo();
```

```
        }
        protected abstract void masInfo();
}
class ProfesorFP extends ProfesorBase {
        @Override
        protected void masInfo(){
                System.out.println("Soy profesor especialista en FP");
        }
}
public class ProfesorApp{
        public static void main(String[] args) {
                ProfesorFP _profesor=new ProcesorFP();
                _profesor.setNombre("Paco");
                _profesor.identificarse();
        }
}
```

5.3. Interfaz vs. Clase Abstracta

Las principales diferencias entre interfaces y clases abstractas son:

- Una interfaz no puede implementar ningún método, al contrario que una clase abstracta (Un interfaz no puede implementar ningún método)
- Una interfaz puede implementar varios interfaces y solamente heredar de una clase (Una clase puede implementar n interfaces, pero solo una clase).
- Una interfaz puede extender otras interfaces. Esta nueva interfaz contendrá los métodos y constantes definidas en las interfaces que extienda, más todos aquellos métodos que se deseen añadir.
- Un interfaz no forma parte de la jerarquía de clases. Clases dispares pueden implementar el mismo interfaz.
- El objetivo de un método abstracto es forzar una interfaz (API) pero no una implementación.

Tipo	Class	Abstract class	Interface
Herencia	extends (simple)	extends (simple)	implements (multiple)
Instanciable	Si	No	No
Implementa	Todos métodos	Algún método	Ningún método
Datos como atributos	Se permiten	Se permiten	No se permiten

Generalización de un algoritmo

- ¿Para qué necesitamos las interfaces si ya tenemos las clases abstractas?

  Como ya hemos comentado Java no permite la herencia múltiple.

  Pensemos en qué puede pasar con la herencia múltiple: una clase que extiende a dos clases en las que se define en ambas un método con la misma signatura.

- ¿Qué ocurre cuando a la subclase se le envíe ese mensaje? ¿Cuál de los dos códigos debe ejecutar?

  Con las interfaces no ocurre esto. Si una clase implementa a dos interfaces y en esas dos interfaces se define el mismo método no hay ningún conflicto ya que no está heredando código sino la responsabilidad de implementar ese método.

Las interfaces se pueden usar para exigir requisitos a la hora de hacer algoritmos genéricos (que puedan trabajar con cualquier tipo de objetos siempre que cumplan los requisitos exigidos en la interface).

Vamos a suponer que tenemos un algoritmo complejo que puede ser interesante para otros usuarios. Supongamos que ese algoritmo trabaja con un determinado tipo de dato.

Vamos a suponer (sin pérdida de generalidad) que este algoritmo super-complejo que deseamos compartir con el resto de usuarios es el siguiente (en la realidad puede tratarse de algoritmos de ordenación).

```
class Utilidades
        public static int maximo ( int a , int b ) f
                int val = a ;
                i f ( b > a )
                        v a l = b ;
                return val ;
        }
}
```

Este algoritmo devuelve el mayor de dos valores enteros.

¿Sería posible una generalización directa de ese código para que cumpla su propósito (obtener el mayor de dos elementos) del siguiente modo?

```
class Utilidades {
      publicstatic Object maximo ( Object a , Object b ) {
            Objectval = a ;
            i f ( b > a )
                  v a l = b ;
            return val ;
      }
}
```

En este código hacemos uso del hecho de que cualquier objeto es de tipo Object.

La respuesta es NO

Ese código no es correcto ya que sobre las referencias no está definida la operación >.

Para hacer nuestro super-algoritmo general tendremos que comenzar por buscar cuál es la operación clave que se realiza sobre los datos:

```
class Utilidades {
      public static int maximo ( int a , int b ) {
            int val = a ;
            i f ( b > a )
                  v a l = b ;
```

```
            return val ;
        }
    }
```

El segundo paso consiste en definir esta operación en una interface:

```
    public interface Compara{
        public boolean mayorQue ( Compara c ) ;
    }
```

El tercer paso consiste en exigir que a nuestro super-algoritmo se le pasen dos objetos del tipo Compara:

```
    class Utilidades{
        public static Compara maximo ( Compara a , Compara b ) {
            Compara val = a ;
            i f ( b.mayorQue( a ) )
                val = b ;
            return val ;
        }
    }
```

Ya puedo ofrecer mi super-algoritmo junto con la interface.

Ahora adopto la postura de un usuario que desea usar el algoritmo (recordemos que se trata de un algoritmo útil y que se supone que ha sido probado exhaustivamente) con sus tipos de datos.

Supongamos que tengo la siguiente jerarquía:

```
    abstract class FiguraCerrada{
        abstract public void pintar ( ) ;
        abstract public double area ( ) ;
    }
    class Rectangulo extends FiguraCerrada{
        // At r i b u t o s
        // M etodos
        public void pintar ( ) {}
        public double area ( ) {}
    }
    class Circulo extends FiguraCerrada{
        // At r i b u t o s
        // Métodos
        public void pintar ( ) {}
        public double area ( ) {}
    }
```

¿Qué debo hacer si quiero usar ese algoritmo para determinar qué figura es mayor en función de su área?

Hacer que en mi jerarquía se implemente la interface Compara

¿En esta jerarquía dónde se podría implementar esa interface?

En la clase FiguraCerrada

```
abstract class FiguraCerrada implements Compara{
      abstract public void pintar ( ) ;
      abstract public double area ( ) ;
      public boolean mayorQue ( Compara o){
            FiguraCerrada f = ( FiguraCerrada ) o ;
            return ( area ()>f.area ( ) ) ;
      }
}
```

Con esta modificación, el siguiente código es válido:

. . .

Rectangulo r = new Rectangulo (10 , 20) ;

Circulo c = new Circulo (20) ;

FiguraCerrada f = (FiguraCerrada) (Utilidades . maximo (r , c)) ;

// Ahora f e s una r e f e r e n c i a a l a f i g u r a con mas a r e a

. . .

5.4. Clases, Subclases, Abstractas e Interfaces

Se hace una clase que no hereda de nadie cuando la clase no pase la prueba de Es-Un.

Se hace una subclase cuando necesitemos hacer una especilización de la superclase mediante sobreescritura o añadiendo nuevos métodos.

Se hace una clase abstracta cuando se quiera definir un grupo genérico de clases y además se tengan algunos métodos implementados que reutilizar. También cuando no queramos que nadie instancie dicha clase.

Se hace un interfaz cuando queramos definir un grupo genérico de clases y no tengamos métodos implementados que reutilizar. O cuando nos veamos forzados por la falta de herencia múltiple en Java.

6. Miembros de una clase

6.1. Miembros de clase o miembros estáticos (static) de una clase

En Java no existen variables globales, por lo tanto, si queremos utilizar una variable única y que puedan utilizar todos los objetos de una clase deberemos de declararla estática (static).

A diferencia de los miembros normales o **miembros de instancia**, los **miembros de clase** tienen la cláusula *static* y todos los objetos de la misma clase compartirán dichos miembros.

Ejemplo de miembros estáticos:

```
public class Cohete {
    private static int numcohetes = 0;
    public Cohete() {
        numcohetes++;
    }
    public static int getCohetes()     {
        return numcohetes;
```

```
    }
}
public class testEstaticos {
    public static void main (String[] args)     {
        Cohete c1 = new Cohete();
        Cohete c2 = new Cohete();
        Cohete c3 = new Cohete();
        System.out.println(c1.getCohetes());
        System.out.println(c3.getCohetes());
c3 = c2; //new Cohete();
System.out.println(c3.getCohetes());
//        System.out.println(Cohete.getCohetes());

    }
}
```

La variable numcohetes se inicializa a 0 una vez, la primera, luego se va incrementando.

Recuerda, los **miembros o atributos de instancia** son aquellos que no son static.

Métodos de instancia y de clase

Los métodos de una clase son una abstracción del comportamiento de la misma.

6.2. Métodos de instancia

Son los métodos comunes.

Cada instancia u objeto tendrá sus propios métodos independientes del mismo método de otro objeto de la misma clase. Ejemplo:

```
public class Cuadrado extends Figura {
       private int lado;
       Cuadrado(int l) {  this.lado = l; }
       public int getArea() {return lado*lado;} //método de instancia
}
```

Los métodos de instancia pueden acceder a los miembros de instancia y también a los miembros de clase(static). Ejemplo:

```
public class test {
       private static int var;
       private int var2;
       public void prueba() {var =3; var2=5;}
}
```

6.3. Métodos de clase o estáticos

Un método por clase, son comunes para una clase.

Los métodos de clase cumplen las siguientes reglas:

1. **Los métodos static no tienen referencia this.**

2. **Un método static no puede acceder a miembros que no sean static.** Un método no static puede acceder a miembros static y no static.

Ejemplo:

```
public class Test {
        public int dato;
        public static int datoEstatico;
        public void metodoInstancia() {
              datoEstatico++;
              this.dato++;
        }
        public static void metodoEstatico() {
          //   this.datoEstatico++;//error: los métodos static no tienen referencia this
            dato++;       //error:los métodos static no puede acceder a miembros no estáticos
        }
        public static void main (String[] args)     {
            dato++; //error:un método static no puede acceder a miembros no static
            datoestatico++;
            metodoEstatico();
            metodoInstancia();//error: un método static no puede acceder a miembros que no sean static
        }
}
```

Un ejemplo de métodos de clase son las funciones de la librería java.lang.Math las cuales pueden ser llamadas anteponiendo el nombre de la clase Math. Ejemplo:

```
Math.cos(angulo);
```

La clase Math tiene muchas funciones trigonométricas y de otro tipo, algunos métodos son:

Método	**Descripción**
static int abs(int a) static long abs(long a) static double abs(double a) static float abs(float a)	Devuelve el valor absoluto del parámetro pasado.
static int max(int a, int b) static long max(long a, long b) static double max(double a, double b) static float max(float a, float b)	Devuelve el mayor de los valores a ó b.
static int min(int a, int b) static long min(long a, long b)	Devuelve el menor de los valores a ó b.

static double min(double a, double b) static float min(float a, float b)	
static double pow(double a, double b)	Potencia de un número. Devuelve el valor de a elevado a b.
static double random()	Devuelve un número double aleatorio entre cero y uno(este no incluido)
static int round(float a) static long round (double a)	Redondea el valor de a al entero más cercano.

7. Paso de parámetros

La mejor forma para llevar a cabo la comunicación ente subprogramas, es el paso de parámetros.

La correspondencia entre **parámetros formales** y **reales** se suele realizar por la posición que ocupan (orden de declaración) y de izquierda a derecha. Para que se pueda realizar esta asociación, tiene que haber el mismo número de parámetros formales y reales, y con el mismo tipo.

Real Funcion (ent x:entero, ent y:real) //**parámetros formales**

//main

Var a:real

a<-5.5

Escribir Funcion (3, a+1) //**parámetros reales**

Escribir a

Según se usen para meter datos o para obtener resultados, los parámetros pueden ser de **entrada, de salida o de entrada/salida**.

Francesco Cesarini (@FrancescoC) twitteó a las 0:00 a. m. on vie., jun. 03, 2016: Really good example of pass by reference vs pass by value. (via Corrado Santoro). https://t.co/ALBS1YYKy2 (https://twitter.com/FrancescoC/status/738490412441145344?s=03)

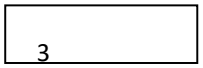
https://twitter.com/i/status/738490412441145344

7.1. Paso de parámetros por valor

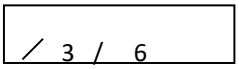
Al hacer la llamada al subprograma se evalúa el valor del **parámetro real** (puede ser una constante, expresión o variable), y ese es el que se asocia, es decir, el que se guarda o asigna al **parámetro formal** asociado. El cual por tanto es una copia del parámetro real.

Se utiliza exclusivamente para parámetros de entrada.

algoritmo Ejemplo
  var a: **entero**
inicio
  a ← 3
  procedimiento (a)
  escribir a
fin



a del PP



x de P

procedimiento procedimiento (entrada x: **entero**)
inicio
  x ← x*2
  escribir x
fin_procedimiento

El valor de a sería 3 y el de x sería 6.

```
public class Valor {
      public static void procedimiento(int param){
        param = param * 2;
      System.out.println(param);
      }
      public static int funcion(int param)
      {
        param = param * 2;
        return param;  //return param * 2;
      }

      public static void main (String[] args)     {
        int argumento;

        argumento = 3;
        procedimiento(argumento);
        System.out.println(argumento);

        System.out.println(función(argumento));
        System.out.println(argumento);
      }
}
```

7.2. Paso de parámetros por referencia

¿Cómo hacemos si queremos que una función o método devuelva más de un valor?

El parámetro formal ha de tener asignada una dirección de memoria en la que se almacena (el parámetro real ha de ser una variable), pero en esa dirección **NO SE GUARDA SU VALOR**, sino que se almacena la dirección de su parámetro real asociado, es decir, el parámetro formal apunta al parámetro real que tiene asociado y cualquier modificación que se efectúe sobre el parámetro formal tendrá una repercusión directa en el parámetro real asociado ya que lo que modificará será el valor almacenado en la dirección que indica el parámetro formal que es la de su parámetro formal asociado.

Para indicar que el tipo de paso de parámetro es por referencia, se utiliza la palabra clave **ent-sal** precediendo al parámetro que se pasa por referencia.

A estos parámetros también se les llama parámetros variables (porque su valor varía), por eso en Pascal se usa la palabra clave Var para indicarlo.

En otros lenguajes como C++, se usan como parámetros punteros para indicar que son direcciones (&, *).

algoritmo Ejemplo

var a : Entero

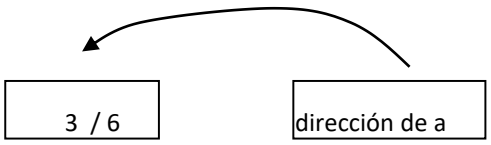
inicio

a ← 3

Procedimiento1(a)

escribir (a)

fin



procedimiento Pocedimiento1(**ent-sal** x:**entero**)

inicio

x ← x *2

escribir (x)

fin_procedimiento

El valor de a y el de x sería 6.

Por valor el parámetro actual no cambia de valor.

Por referencia el parámetro actual puede cambiar. Este ejemplo no pasa por referencia sino por valor.

```
public class Valor {
      public static void P1(Integer param)
      {
        int valor = param.intValue();
        System.out.println(param.intValue());
        valor = valor * 2;
        param = new Integer(valor);
        System.out.println(param.intValue());
      }
      public static void main (String[] args)     {
      {
        Integer a = new Integer(3);//encapasula un int en un objeto
      Integer
        P(a); //Los objetos se pasan por referencia
        System.out.println(a.intValue()); //sale 3
      }
}
```

Las funciones son las únicas que tienen un parámetro exclusivamente de salida y el paso de ese parámetro es por valor.

Lo ideal sería que la clase Integer tuviese un método AsignarValor para hacer lo siguiente y no tener que crear otro objeto.

```
      public static void P1(Integer param)
      {
```

```
            int valor = param.intValue();
            valor = valor * 2;
            param.AsignarValor(valor);
        }
```

Pero como las clases que encapsulan a los tipos primitivos no tienen tales métodos esta forma no sirve.

Los arrays se pasan siempre por referencia o dirección, con los cual los cambios que haga el método sobre el parámetro afectarán al agumento:

```
public class Valor {
      public static void P1(int[] param)
      {
        param[0] = param[0] * 2;
      }
      public static void main (String[] args)     {
      {
        int[] a = {3};  //int a = 3;
        P1(a) //Los arrays se pasan por referencia
        System.out.println(a[0]);
      }
}
```

7.3. Parámetros por referencia en Java

En Java solo hay paso de parámetros por copia. Pero entonces, por qué la gente habla del paso de parámetros por referencia en Java. Veamos en detalle por qué parece que realizamos un paso de parámetros por referencia en Java.

7.3.1. Paso de parámetros por valor

Lo primero que tenemos que ver es que para los datos primitivos en Java se realiza claramente una copia.

```
1. public void metodo(int p) {
2.   p=3;
3. }
4.
5. int p1=2;
6. metodo(p1);
7.
8. System.out.println(p1); //p1 = 2
```

7.3.2. Paso de parámetros "por referencia": referencia de objetos

Pero ahora pasemos a manejar un objeto como parámetro. Lo que sucede al manejar los objetos en Java es que las variables mantienen una referencia al objeto, por lo tanto, cuando pasamos un objeto como parámetro se está realizando una copia de la referencia. Así tenemos dos variables diferentes apuntando al mismo objeto.

Creamos una clase básica llamada MiClase:

```
public class MiClase {
 public int valor;
}
```

Y ahora un método que modifica ese valor:

```
public static void metodo_referencia(MiClase m) {
  m.valor =m.valor*2;
}
```

Veamos como se pasa por referencia, aunque parece que sea por valor:

```
MiClase m1 = new MiClase();
m1.valor = 3;
System.out.println(m1.valor); // Devuelve 3
metodo_referencia(m1);
System.out.println(m1.valor); // Devuelve 6
```

7.3.3. Desmitificando el paso de parámetros "por referencia"

Pero para verlo mejor veamos otro caso. Ahora lo que vamos a hacer es crear un nuevo objeto y cambiar la referencia de la variable pasada por copia.

```
public static void metodo_referencia2(MiClase m1) {
  MiClase m2 = new MiClase();
  m1 = m2;
  m1.valor = 3;
}
```

Lo que sucede es que ahora m1, que mantenía una referencia al objeto inicial, pasa a tener una referencia a un nuevo objeto. Por lo tanto, los cambios que hagamos en m1 no afectan ya al objeto inicial.

Si volvemos a realizar la misma secuencia de salida vemos que no hay cambios en el objeto inicial.

```
MiClase m2 = new MiClase();
m2.valor = 2;
System.out.println(m2.valor); // Devuelve 2
metodo_referencia2(m2);
System.out.println(m2.valor); // Devuelve 2
```

Por lo tanto, siempre debemos de tener en cuenta que los parámetros en Java se pasan por valor. Pero que existen las referencias a los objetos y por lo tanto podemos tener la falsa sensación de que hay paso de parámetros por referencia en Java.

8. Métodos recursivos

Se dice que un método o subprograma es recursivo cuando se llama a sí mismo. La recursividad se va a poder usar con problemas que pueden definirse en términos recursivos, es decir, **en término de sí mismo**, como procesos de alguna manera repetitivos.

La recursividad permite definir ciertos algoritmos de forma más sencilla.

La recursividad se trata de evitar siempre que sea posible, es decir, siempre que se pueda solucionar con una iteración, ya que cada vez que un subprograma se llama a sí mismo hay que almacenar en la pila del sistema la

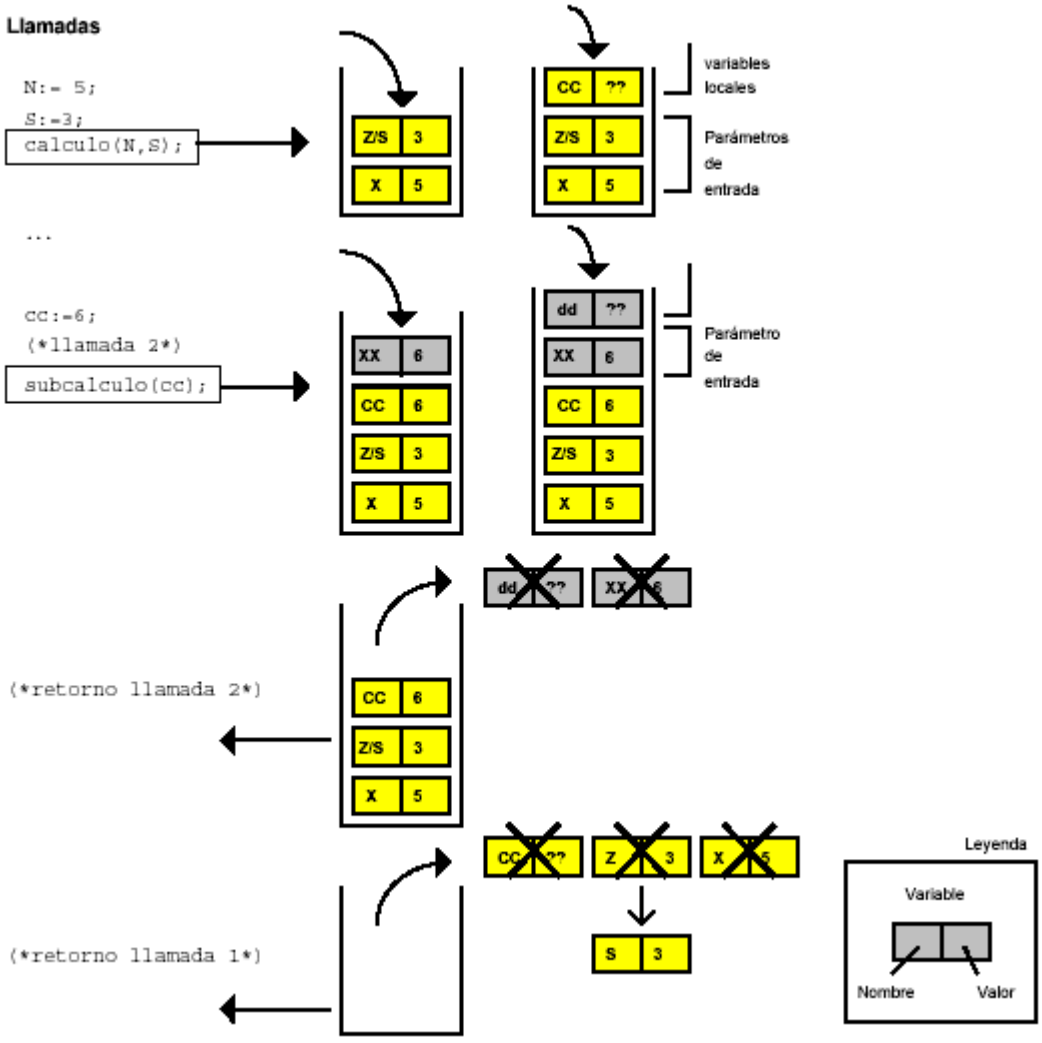
dirección de retorno de ese subprograma, con lo cual si hacemos muchas llamadas recursivamente iremos llenando la pila del sistema, y se desbordara acabando con la memoria.

Inconvenientes de las funciones recursivas:

- no ahorran espacio,
- se ejecutan más despacio que las versiones iterativas,
- es posible que la pila se mezcle con otros datos u otro programa.

Ventajas:

- son más claras y sencillas que algunas iterativas,
- algunos problemas (inteligencia artificial) tienden hacia soluciones recursivas,
- se adapta más al pensamiento humano.



Todo programa recursivo tiene que tener un ***caso base*** o ***condición de parada*** que ponga fin a la recursividad, es decir, que el programa deje de llamarse a sí mismo cuando se cumpla la condición, sino se formaría un bucle infinito.

La fórmula o proceso reducirá la complejidad y nos irá acercando a la solución

Ejemplo:

funcion potencia (base: **entero**; expo:**entero**): **entero**

```
var P: entero
    i: entero
inicio
  p ←1
  desde i = 1 hasta expo
    p ← p* base
  fin_desde
  retorno p
fin
```

```
función potencia (base: entero; expo: entero): entero
var entero temp;
inicio
  si expo == 0 entonces
    temp = 1
  sino
    temp = (base*potencia(base, expo-1)
  retorno temp;
fin_función
```

```
public static int potencia(int base, int expo)
{
  int temp;
  if (expo == 0) { //caso base
      temp = 1;
  }
  else {//reducimos complejidad
     temp = base * potencia(base, expo - 1);
  }
  return temp.;
}
```

Ejemplo:

```
public static long factorial(int n) {
      long f;
      if ( n > 1) {

            f = n * factorial(n-1);
      }
      else
            f = 1;
      return f;
}
```

Cuando únicamente existe una llamada, se denomina *recursividad lineal*, si existen varias es una *recursividad múltiple*. Con esta existe el riesgo de que muchas de las llamadas se repitan, por lo que la versión recursiva del algoritmo no resulta recomendable. Ejemplo:

```
public static int fibonacci(int n)
{
      int fib;
      if (n == 0 || n == 1)
```

```
                fib = n;
            else
                fib = fibonacci(n-1) + fibonacci(n-2);
            return fib;
        }
```

Anexo A. Implements

El extends es para declarar una relación de herencia. Esto puede ser entre dos clases (public class ClassA extends ClassB) o entre interfaces (public interface InterfaceA extends InterfaceB).

De esta manera, la clase (o interfaz) B tendrá una relación "es un" con la clase A. Ejemplo:

```
public class Perro extends Animal
```

En este caso podemos decir que un Perro es un Animal y podremos usar un Perro en cualquier lugar donde se espere un animal.

El implements por otro lado se usa cuando una clase debe tener los métodos declarados en una interfaz.

Por ejemplo:

```
public interface Parlante {
	public void decirAlgo();
}
```

Luego:

```
public class Persona implements Parlante{
	public void decirAlgo(){
		System.out.println("Hola");
	}
}

public class Perro implements Parlante {
	public void decirAlgo(){
		System.out.println("Guau");
	}
}
```

El implements obliga a la clase a (justamente) implementar los métodos que estaban definidos en la interfaz. Como se puede ver cada clase implementa la interfaz a su manera.

Además se pueden hacer cosas como

```
	Parlante p = new Perro();
	Parlante p1 = new Persona();
```

y hacer

```
p.decirAlgo();
p1.decirAlgo();
```

y cada uno se comportará de manera distinta aunque hayas declarado ambos como parlantes. De la misma manera que antes en cada lugar donde se espere un Parlante se podrá pasar un objeto de cualquier clase que implemente dicha interfaz.

Las interfaces se utilizan para definir comportamiento.

Cuando entiendas esto fíjate lo que es una **clase abstracta**, que es algo a mitad de camino entre ambas cosas.