ATRIBUTOS DE INSTANCIA: Variables asociadas a cada objeto de la clase
RUBY: @atributo_ de_instancia = valor. Se declaran en el metodo initialize.
JAVA: (visibilidad) (tipo) deinstancia = defecto; Se declaran al crear la clase.
ATRIBUTOS DE CLASE: Son variables asociadas a la propia clase y no a cada instancia. Seran globales a todas las instancias. Existen de forma única. Cuando su valor cambia lo hace en todas las instancias. Se suele usar para evitar el uso de números mágicos o en constantes o contadores.
RUBY: @@declase = valor. Se declaran en la clase(fuera de metodos instanc)
JAVA: (vis) static (tipo) DECLASE = valor. Se declaran al crear la clase.
METODOS DE INSTANCIA: Funciones asociadas a los objetos de la clase. Desde un metodo de instancia se puede acceder a los atributos de instancia de dicho objeto.

RUBY:
```
def metodo_instancia
    return @deinstancia
end
```
JAVA:
```
(visibilidad) (tipo) metodoInstancia(){
        return deinstancia;
}
```

METODOS DE CLASE: Funciones asociadas a la propia clase. Habitual que accedan/actualicen atrib de clase. No se puede acceder a atributos/metodos de instancia desde un metodo de clase.

RUBY:
```
def self.metodo_clase
    @@declase+=1
    return @@declase
end
```
JAVA:
```
(visibilidad) static (tipo) metodoClase){
        DECLASE++;
        return DECLASE;
}
```

PARTICULARIDAD RUBY: Dos tipos de atributos de clase
@@atributo_de_clase @atributo_de_instancia_de_clase
Los atributos de clase son accesibles desde ambito de instancia, los de instancia de clase no. Los atributos de clase se comparten con las subclases mientras que los de instancia de clase no. ¿Diferencia entre @atributo_ de_instancia y @atributo_de_instancia_de_clase? Depende del ámbito.
ÁMBITO: En una clase cualquier punto dentro de un metodo de instancia esta en ambito de instancia, lo demás esta en ambito de clase. (Esto también es así en Java)

```
class Clase
    @@atributo_de_clase
    @atributo_de_instancia_de_clase
    def metodo_instancia
        @atributo_ de_instancia
    end
end
```

VARIABLES LOCALES: Sin @.
CONSTANTES: final en Java y en mayúscula en Ruby

PSEUDOVARIABLES: this en Java, self en Ruby. Hacen referencia al propio objeto o a la clase (Depende del ambito). Desde ambito de clase no se puede usar this o self para llamar a atributos.
ESPECIFICADORES DE ACCESO
JAVA:

Visible	Mismo paquete		Otro paquete
	Clase	Otra	Otra
private	X		
package	X	X	
protected	X	X	
public	X	X	X

PRIVADO: DIFERENCIAS ENTRE JAVA Y RUBY
En Java se puede acceder a atributos y metodos privados desde una instancia, ambito de clase o ambito de instancia mientras que en ruby no se puede hacer nada de eso.

RUBY:

Visible	Mismo objeto	Clase	Otra
private	X		
protected	X		
public	X	X	X

Para dar visibilidad en Ruby se coloca el especificador y todos los metodos que vayan delante tendran la visibilidad especificada. Tambien se puede poner especificador :nombre_metodo_instancia para los de instancia o especificador_class_method :nombre_metodo_clase para los de clase.
CONSTRUCTORES:
Se deben inicializar todos los atributos de instancia.
RUBY:
El constructor de ruby es el metodo especial llamado initialize. Metodo privado de instancia que se llama automaticamente al llamar al metodo de clase new. Se ocupa de la **creación** e inicializacion de atributos de instancia. No se puede sobrecargar initialize (ni ningún metodo). Aun asi se pueden crear varios constructores, creando varios metodos de clase que funcionen como constructores o haciendo que initialize admita un número variable de parametros.

```
class Clase
    def initialize(parametro1, parametro2) #Creamos e inicializamos atrib.de.inst.
        @atributo1 = parametro1
        @atributo2 = parametro2
    end
    def self.new_2parametros(parametro1, parametro2)
        new ( parametro1, parametro2) #Llama a initialize con dos parametros
    end
    def self.new_1parametros1(parametro1)
        new ( parametro1, 0) #Llama a initialize con un parametro y el otro a 0
    end
    private_class_method :new #Obligamos a usar los "constructores nuevos"
end
```

JAVA:
Mismo nombre que la clase. Se usan para asegurar la inicializacion de los atributos. Puede haber varios. Se puede reutilizar un contructor desde otro constructor. Para construir el objeto se antepone la palabra new al nombre de la clase. Si no se crea ningun constructor existe uno por defecto sin parametros.

```
class Clase {
    atributo1;
    atributo2;
   NM = valor_defecto_atributo2;
    Clase(parametro1, parametro2) { //Constructor 2 parametros
       this.atributo1 = parametro1;
       this.atributo2 = parametro2;
    }
    Clase(parametro1) { //Constructor 1 parametro
        this(parametro1, NM);
    }
    Clase(Clase otro) { //Constructor copia
       this(otro.atributo1,  otro.atributo2);
    }
}
```

CONSULTORES Y MODIFICADORES
JAVA: getAtributo{return atributo},void setAtributo(par){atributo = par}
RUBY: attr_reader :atributoinstancia #consultor
attr_writer :atributoinstancia #modif attr_accesor :atributoinstancia #ambos

AGRUPACION: module en Ruby y package en Java (no existen subpaquetes en Java)
Para referenciar archivos en Ruby se usa require_relative(archivo propio) o require (del lenguaje)
HERENCIA: Establece una relacion es-un. La clase de la que se deriva es la clase padre, la clase derivada es la clase hija o subclase. Hay transitividad. La clase hija hereda todo el codigo. Los atributos privados heredados no son accesibles desde la clase hija (Esto no quiere decir que no puedas obtenerlo, por ejemplo, a traves de un getter obtienes una referencia al atributo permitiendo tanto verlo como modificarlo).
RUBY: class Hija < Padre #La clase Hija hereda de la clase Padre
JAVA: class Hija extends Padre //La clase Hija hereda de la clase Padre
REDEFINICIÓN (O SOBREESCRITURA) DE METODOS (VS SOBRECARGA):
Un metodo se redefine cuando una clase actua distinto que la clase de la que ha heredado(Se anula la implementacion heredada). Por ejemplo, si tenemos la clase Persona con el metodo hablar y una clase Profesor que hereda de Persona de manera que el metodo hablar ahora es distinto, lo redefinimos para que hable como un profesor. Se puede usar el codigo heredado en la redefinición de manera que el metodo sobreescrito extiende al del padre. Para esto usamos super que permite ejecutar la implementación del padre del método que se esta redefiniendo. Super en Ruby y Java.
RUBY: Solo permite acceder en la clase padre a la implementacion del metodo que se esta redefiniendo. Si se utiliza sin argumentos se pasan automaticamente los mismos que esta recibiendo el metodo redefinido. En initialize es igual que en otro metodo. Al crear un método con el mismo nombre que en la superclase se produce redefinición ya que no hay sobrecarga.

```
class  Padre
 def initialize(par1)
    @atrib1 = par1
 end
 def metodo1
    puts "Soy padre con #{@atrib1}"
 end
end
```

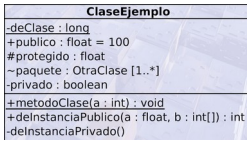
```
class Hija < Padre
    def initialize(par1, par2)
        super(par1) #Llama initialize padre
        @atrib2 = par2
    end
    def metodo1 #Redefinicion
        super #
        puts "y hija con #{@atrib2}"
    end
```

Si no realizamos la llamada a super en el initialize de la clase Hija los atributos de instancia de la clase Padre no existiran en Hija y dara error al hacer llamada a metodos que los usen (si el metodo no realiza cambios y simplemente hace puts no dara error pero no se verá nada).
JAVA: Permite acceder a la implementacion de cualquier metodo del padre. No es recomendable usarlo para llamar a metodos distintos del que se esta redefiniendo. En el constructor permite referenciar al constructor de la clase padre (si tiene varios en funcionen de los parametros), pero debe aparecer en la primera linea del constructor de la clase derivada. No se pueden redefinir metodos declarados como final ni metodos privados del padre. Es aconsejable usar @Override ya que el compilador te avisa si estas sobrecargando en lugar de redefiniendo. Se permite ampliar la accesibilidad del especificador de acceso (No disminuir).

```
class  Padre {
 atrib1;
 Padre (par1) {
    atrib1 = par1;
 }
 String metodo1{
    return  "Soy padre con " + atrib1
 }
}
```

```
class Hija extends Padre {
    atrib2;
    Hija(par1, par2) {
        super(par1); #Llama al constructor de Padre
        atrib2 = par2
    }
    String metodo1{ #Redefinicion
        super();
        return  "Soy hija con " + atrib2
    }
}
```

En la sobrecarga permitimos que un metodo actue de varias maneras en funcion de los parametros o tipo del parametro que se le pasa al metodo.



UML:
Asociaciones que generan atributos de referencia:
Asociacion --------
PARTE ----- (tipo de asociacion especial) TODO
Agregacion ----<> En el TODO cardinalidad 0..1
Composicion ----◆ En el TODO cardinalidad 1
Dependencia ----→ No genera atributo de referencia
Cardinalidad. 1 por defecto. n*… n a muchos. Si es 0 significa que puede haber o no asociacion.