

# Elementos de Agrupación: Paquetes y Módulos




Programación y Diseño Orientado a Objetos (PD00)

## Tabla de Contenidos

- 1. [Paquetes en Java](#paquetes-en-java)
- 2. [Módulos en Ruby](#módulos-en-ruby)
- 3. [Proyectos Ruby con Múltiples Archivos](#proyectos-ruby-con-múltiples-archivos-require\_relative)

## Introducción

Los **paquetes** y **módulos** son mecanismos esenciales en POO para:

-  Agrupar elementos relacionados (clases, constantes, funciones)
-  Gestionar el espacio de nombres
-  Evitar conflictos de nombres

## Paquetes en Java

Los paquetes se utilizan para agrupar clases y funcionan como un **espacio de nombres** jerárquico que, sin embargo, no implica subpaquetes reales a nivel de lenguaje.

## Características Clave

Característica	Descripción
Espacio de Nombres	Permiten tener varias clases con el mismo nombre en paquetes distintos
Visibilidad de Paquete	Nivel de acceso especial (por defecto) que restringe visibilidad al mismo paquete
Estructura en Disco	Un paquete se corresponde con una <b>carpeta</b> del sistema de ficheros
Independencia	❌ <code>view.gui</code> y <code>view.tui</code> son independientes de <code>view</code>

# Uso y Sintaxis

Operación	Sintaxis	Descripción
Declaración	package nombre;	Se añade al inicio del archivo .java
Importación	import paquete.Clase;	Para usar clases de otro paquete

## Ejemplo en Java

```
// Archivo Vista.java
package view; // Declaración del paquete
public interface Vista {
    // ...
}

// Archivo VistaGrafica.java
package view.gui; // PAQUETE DISTINTO
import view.Vista; // Se importa la clase/interfaz del paquete 'view'
class VistaGrafica implements Vista {
    // ...
}
```

# Módulos en Ruby

Los módulos son **más flexibles** que los paquetes de Java, ya que agrupan una gran variedad de elementos (clases, constantes, funciones, otros módulos).

## Características Clave

Característica	Descripción
Espacio de Nombres	Definen un contexto para evitar colisiones de nombres
Jerarquía	☒ A diferencia de Java, <b>sí puede haber módulos dentro de módulos</b>
Inclusión (include)	Se puede copiar literalmente el contenido de un módulo dentro de una clase (mixins)
Acceso	Se accede encadenando nombres con ::

# Uso y Sintaxis

Operación	Sintaxis	Descripción
Definición	module Nombre ... end	Similar a clases
Acceso Directo	Modulo::Clase.new	Encadenar con ::

Inclusión	include Modulo	Copiar contenido en la clase
-----------	----------------	------------------------------

## Ejemplo: Módulos Anidados y Acceso

```
module Externo
  class A
    end
    module Interno # Módulo anidado
      class B
        end
      end
    end
  end
end
module Test
  def test
    puts "Testeando"
  end
end
class C
  include Test # Copia el método 'test' a la clase C
end
# Uso de módulos anidados
# $a = Externo::A.new
# $b = Externo::Interno::B.new
# Uso del mixin (módulo incluido)
# c = C.new
# c.test
```

---

## Proyectos Ruby con Múltiples Archivos (require\_relative)

Dado que Ruby es un lenguaje **interpretado** y no compila todos los archivos de antemano, debe indicársele explícitamente qué archivos cargar antes de que sus clases sean utilizadas.

### El Problema

☒ Si la clase Persona en persona.rb usa la clase Cosa de cosa.rb, y no se le indica a Ruby que cargue cosa.rb primero, se generará un error: `NameError: uninitialized constant`.

### Solución

Se usan las instrucciones `require` o `require_relative`:

Instrucción	Uso	Ejemplo
require	Cargar archivos de la biblioteca estándar	require 'date'
require_relative	Cargar archivos propios del proyecto (ruta relativa)	require_relative 'cosa'

## Criterio de Uso (Buenas Prácticas)

☒ **Regla:** Cuando en un archivo se menciona el nombre de una clase definida en otro archivo, se debe añadir un `require_relative` al inicio del archivo que contiene la referencia.

## Ejemplo de `require_relative` Correcto

```
# principal.rb
require_relative 'cosa' # Necesario porque 'Cosa' se usa en la línea 4
require_relative 'persona' # Necesario porque 'Persona' se usa en la línea 5
mochila = Cosa.new("Mochila")
juan = Persona.new("Juan")
juan.otra_cosa_mas(mochila)
```

## ✗ Mala Práctica

**NO** añadir `require_relative` de todos los archivos en todos los archivos, ya que esto puede generar:

- Errores por **\*\*dependencia circular\*\***
- Carga de código **\*\*innecesaria\*\***

## Tabla Comparativa: Java vs Ruby

Aspecto	Java (Paquetes)	Ruby (Módulos)
Propósito	Agrupar clases	Agrupar clases, constantes, funciones, módulos
Jerarquía	Apariencia jerárquica (no real)	☒ Jerarquía real (módulos dentro de módulos)
Estructura de disco	Carpetas	Archivos independientes
Acceso	import paquete.Clase;	Modulo::Clase o include Modulo
Mixins	☒ No soportado	☒ include para copiar contenido

Carga de archivos	Automática (compilación)	Manual (require_relative)
-------------------	--------------------------	---------------------------

---

## Buenas Prácticas

Práctica	Java	Ruby	Prioridad
Organización lógica	Paquetes por funcionalidad	Módulos por dominio	☒ Crítico
Nombres descriptivos	com.empresa.proyecto.modulo	Empresa::Proyecto::Modulo	☒ Crítico
Evitar dependencias circulares	Diseño modular	Usar require_relative con cuidado	☒ Importante
Visibilidad mínima	package-private cuando sea posible	Métodos privados	☒ Recomendado

---

Adam Bourbahr Romero ~ PDOO