JAVA-10 - Programación orientada a objetos - Avanzado

Objetivos de aprendizaje:

Al final de la sesión, seré capaz de:

- Usar paquetes Java para organizar proyectos.
- Explicar los modificadores de acceso Java.
- Explicar y usar la encapsulación.
- Declarar, modificar e iterar colecciones de objetos.
- Describir y usar excepciones Java.
- Usa bloques try/catch.

Ordenando nuestro proyecto Java. Uso de los paquetes en Java (packages)

Watch Video At: https://youtu.be/hyJJLXetKCY

Los paquetes en Java (packages) son la forma en la que Java nos permite agrupar de alguna manera lógica los componentes de nuestra aplicación que estén relacionados entre sí.

Los paquetes permiten poner en su interior casi cualquier cosa como: clases, interfaces, archivos de texto, entre otros. De este modo, los paquetes en Java ayudan a darle una buena organización a la aplicación ya que permiten modularizar o categorizar las diferentes estructuras que componen nuestro software.

Los paquetes en Java, adicionalmente al orden que nos permite darle a la aplicación, también nos brindan un nivel adicional de seguridad para nuestras clases, métodos o interfaces, pues como veremos más adelante podremos especificar si una clase o interfaz en particular es accesible por todos los componentes del software (sin importar el paquete) o si en realidad es solo accesible por las clases que estén en el mismo paquete que ésta.

Ahora veamos cómo crear paquetes en Java

¿Cómo crear paquetes en Java?

Ya sabemos para qué sirven los paquetes en Java y sus características principales. Vamos ahora a aprender cómo usarlos y qué cambios generan estos en la estructura de nuestro proyecto.

Para declarar un paquete en Java se hace uso de la palabra reservada "package" seguido de la "ruta" del paquete, como se muestra a continuación.

```
package ruta.del.paquete;
```

Hay varias cosas que clarificar aquí. Como verás, la sintaxis es bastante simple y comprensible pero hay algunos detalles a tener en cuenta, veamos.

Tips o prácticas para crear paquetes en Java

El paquete en Java se declara antes que cualquier otra cosa:

La declaración del paquete debe estar al principio del archivo Java, es decir, es la primera línea que se debe ver en nuestro código o archivo .java. Primero se declara el paquete, y luego podremos poner los imports y luego las clases, interfaces, métodos, etc.

- **Cada punto en la ruta del paquete es una nueva carpeta:**

  Cuando se escribe la ruta del paquete en Java, se pueden especificar una ruta compleja usando el punto ".", por ejemplo en el código anterior, nuestra clase, interfaz o archivo estará ubicado en una carpeta llamada "paquete" la cual a su vez está en la carpeta "del" y esta también se encuentra dentro de una carpeta llamada "ruta". De este modo si queremos por ejemplo que una clase quede al interior de una carpeta llamada "mi_paquete" y ésta al interior de una carpeta llamada "otro_paquete" pondríamos:

```
package otro_paquete.mi_paquete; /*Se usa el punto para separar cada carpeta equivale a la ruta otro_paquete/mi_paquete dentro del proyecto*/ public class mi_clase { }
```

Prácticas recomendadas para nombrar paquetes en Java

También hay varias buenas prácticas y recomendaciones para crear y declarar paquetes en Java. En generarl, los paquetes en java se declaran siempre en minúsculas y en caso de ser necesario las palabras se separan usando un guión bajo "_".

Si no se declara un paquete (paquete por defecto):

- Si decidimos no declarar un paquete para nuestra clase, ésta quedará en un paquete que se conoce como paquete por defecto (default package), en este paquete estarán todas las clases que no tengan un paquete declarado. Aunque esto no genera errores de compilación ni nada parecido, siempre es recomendable declarar un paquete a cada componente de nuestro programa Java para poder darle diferentes niveles de seguridad o acceso a dichos componentes y mantener todo ordenado.

Al momento de declarar el paquete en Java:

Es común usar la primera letra en mayúscula cuando se declara una clase, pues bien, cuando se declaran paquetes es común que todas la letras estén en minúscula y en caso de ser varias palabras separarlas por un guion bajo "_" por ejemplo "mi_paquete" es adecuado mientras que "MiPaquete" aunque no es incorrecto, no es una buena práctica.

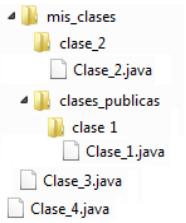
- En estos momentos ya tenemos claro qué es y para qué sirve un paquete en Java, también sabemos cómo se declaran los paquetes en Java y probablemente tendremos una noción de los cambios que estos generan al interior de la estructura de nuestro proyecto. A continuación veremos un ejemplo que consistirá de cuatro clases diferentes en ubicaciones (paquetes) también diferentes y luego veremos cómo queda la estructura de nuestro proyecto

Ejemplo de paquetes en Java

A continuación pondré la declaración de cuatro clases diferentes en paquetes diferentes, cada clase se llamará "Clase_1", "Clase_2", "Clase_3" y "Clase_4" respectivamente. Voy a jugar un poco con las carpetas donde quedará alojada cada una de las clases para que afiances un poco más el concepto y comprendas bien el funcionamiento de los paquetes en Java y quizá soluciones alguna duda. Por tal motivo, te invito a intentar descifrar la ubicación exacta de cada clase al interior del proyecto según la declaración de su respectivo paquete y ver que sí hayas entendido adecuadamente el tema. Veamos:

- **Clase número uno:**
  `package mis_clases.clases_publicas.clase_1; public class Clase_1 { }`
- **Clase número dos:**
  `package mis_clases.clase_2; class Clase_2 { }`
- **Clase número tres:**
  `package mis_clases; class Clase_3 { }`
- **Clase número cuatro:**
  `class Clase_4 { }`

En la imagen de abajo se puede apreciar la estructura final obtenida tras la declaración de los paquetes de cada una de las cuatro clases de Java. Veamos con detalle esto para quedar claros.



La clase número uno fue declarada en el paquete "mis_clases. clases_publicas.clase_1" por lo tanto quedará al interior de una carpeta llamada "clase_1" la cual estará al interior de otra carpeta llamada "clases_publicas" y esta a su vez estará dentro de la carpeta llamada "mis_clases". La clase número dos se le declaró el paquete "mis_clases.clase_2" por lo tanto ha quedado al interior de una carpeta llamada "clase_2" la cual a su vez está al interior de la carpeta "mis_clases".
A la clase número tres se le impuso el paquete "mis_clases", esto indica que esta estará al interior de la carpeta "mis_clases". Por último, la clase número cuatro no tiene declarado ningún paquete por lo tanto el paquete asignado será el paquete por defecto, en otras palabras, esta clase se mantendrá siempre en la carpeta raíz del proyecto (en netbeans y eclipse la carpeta raíz se llama src).

Concluyendo el ejemplo de los paquetes en Java (packages)

Concluyendo finalmente con el ejemplo, tenemos que las tres primeras clases se encuentran al interior de la carpeta "mis_clases", la cuarta clase no se encuentra allí dado que se encuentra en la carpeta raíz del proyecto. La clase dos además de estar al interior de

"mis_clases" está en un carpeta propia "clase_2" y la clase uno, se encuentra además al interior de la carpeta "clases_publicas" (si lo notaste es la única clase publica) y luego al interior de su propia carpeta "clase_1"

Modificadores de acceso public, protected, default y private en Java. Encapsulamiento en Java

Watch Video At: https://youtu.be/cPcjiwSN1YU

Los *modificadores* de acceso nos introducen al concepto de *encapsulamiento*. El encapsulamiento busca de alguna forma controlar el *acceso a los datos* que conforman un objeto o instancia, de este modo podríamos decir que una clase y por ende sus objetos que hacen uso de modificadores de acceso (especialmente privados) son objetos encapsulados.

Los modificadores de acceso permiten dar un nivel de seguridad mayor a nuestras aplicaciones restringiendo el acceso a diferentes atributos, métodos, constructores asegurándonos que el usuario deba seguir una "ruta" especificada por nosotros para acceder a la información.

Es muy posible que nuestras aplicaciones vayan a ser usadas por otros programadores o usuarios con cierto nivel de experiencia; haciendo uso de los modificadores de acceso podremos asegurarnos de que un valor no será modificado incorrectamente por parte de otro programador o usuario. Generalmente el acceso a los atributos se consigue por medio de los métodos get y set, pues es estrictamente necesario que los atributos de una clase sean privados.

Nota: Siempre se recomienda que los atributos de una clase sean privados y por tanto cada atributo debe tener sus propios métodos get y set para obtener y establecer respectivamente el valor del atributo.

Nota 2: Siempre que se use una clase de otro paquete, se debe importar usando import. Cuando dos clases se encuentran en el mismo paquete no es necesario hacer el import pero esto no significa que se pueda acceder a sus componentes directamente.

Veamos un poco en detalle cada uno de los modificadores de acceso

Modificador de acceso private

El modificador *private* en Java es el más restrictivo de todos, básicamente cualquier elemento de una clase que sea privado puede ser accedido únicamente por la misma clase por nada más. Es decir, si por ejemplo, un atributo es privado solo puede ser accedido por lo métodos o constructores de la misma clase. Ninguna otra clase sin importar la relación que tengan podrá tener acceso a ellos.

```java
package aap.ejemplo1;
public class Ejemplo1
{
    private int atributo1;//Este atributo es privado
    private int contador = 0; //Contador de registro

    //Si un atributo es privado podemos crear método get y set ...
    //... para éste y permitir el acceso a él desde otras instancias

    public void setAtributo1(int valor)
    {
        contador++;//Contador que lleva el registro de ediciones del atributo1
        atributo1 = valor;//Establecemos el valor del atributo
    }

    public int getAtributo1()
    {
        return atributo1;//Retornamos el valor actual del atributo
    }

    //Get para el contador
    public int getContador()
    {
        return contador;
    }
    //Notar que no ponemos un set, pues no nos interesa que el contador pueda se
}
```

En el ejemplo anterior vemos lo que mencioné al comiendo, tenemos un atributo privado y permitimos el acceso a él únicamente por medio de los métodos de get y set, notemos que estos métodos son públicos y por tanto cualquiera puede accederlos. Lo realmente interesante con los métodos get y set es que nos permiten realizar cualquier operación como por ejemplo llevar una cuenta de la veces que se estableció el valor para el atributo permitiéndonos mantener nuestro sistema sin problemas. También debemos notar que debido a que los métodos get y set son propios de la clase no tienen problemas con acceder al atributo directamente.

El modificador por defecto (default)

Java nos da la opción de no usar un modificador de acceso y al no hacerlo, el elemento tendrá un acceso conocido como *default*o acceso por defecto que permite que tanto la propia clase como las clases del mismo paquete accedan a dichos componentes (de aquí la importancia de declararle siempre un paquete a nuestras clases).

```java
package aap.ejemplo2;
public class Ejemplo2
{
    private static int atributo1;//Este atributo es privado
    static int contador = 0; //Contador con acceso por defecto

    public static void setAtributo1(int valor)
    {
        contador++;//Contador que lleva el registro de ediciones del atributo1
        atributo1 = valor;//Establecemos el valor del atributo
    }

    public static int getAtributo1()
    {
        return atributo1;//Retornamos el valor actual del atributo
    }
}
```

```java
package aap.ejemplo2;
public class Ejemplo2_1
{
    public static int getContador()
    {
            return Ejemplo2.contador;//Accedemos directamente al contador desde o
    }
}
```

Modificador de acceso protected

El modificador de acceso *protected* nos permite acceso a los componentes con dicho modificador desde la misma clase, clases del mismo paquete y clases que hereden de ella (incluso en diferentes paquetes). Veamos:

```java
package aap.ejemplo3;

public class Ejemplo3
{
    protected static int atributo1;//Atributo protected
    private static int atributo2; //Atributo privado
    int atributo3;//Atributo por default

    public static int getAtributo2()
    {
        return atributo2;
    }
}
```

```java
package aap.ejemplo3_1;

import aap.ejemplo3.Ejemplo3;//Es necesario importar la clase del ejemplo 3

public class Ejemplo3_1 extends Ejemplo3
{
    public static void main(String[] args)
    {
        //La siguientes dos líneas generan error, pues atributo2 es privado y atr
        //System.out.println(atributo2);
        //System.out.println(atributo3);

        System.out.println(atributo1);//Sí tenemos acceso a atributo1
    }
}
```

Modificador public

El modificador de acceso *public* es el más permisivo de todos, básicamente public es lo contrario a private en todos los aspectos (lógicamente), esto quiere decir que si un componente de una clase es public, tendremos acceso a él desde cualquier clase o instancia sin importar el paquete o procedencia de ésta.

```java
package aap.ejemplo4;

public class Ejemplo4
{
    public static int atributo1; //Atributo publico

    public static void metodo1()
    {
        System.out.println("Método publico");
    }
}
```

```java
package paquete.externo;

import aap.ejemplo4.Ejemplo4; //importamos la clase del ejemplo4

public class ClaseExterna
{
    public static void main(String[] args)
    {
        System.out.println(Ejemplo4.atributo1);
        //Tuvimos Acceso directo por ser publico

        Ejemplo4.metodo1(); //Metodo1 también es publico
    }
}
```

A continuación y ya para finalizar, pondré una pequeña tabla que resume el funcionamiento de los modificadores de acceso en Java.

Modificador	La misma clase	Mismo paquete	Subclase	Otro paquete
private	Sí	No	No	No
default	Sí	Sí	No	No
protected	Sí	Sí	Sí/No	No
public	Sí	Sí	Sí	Sí

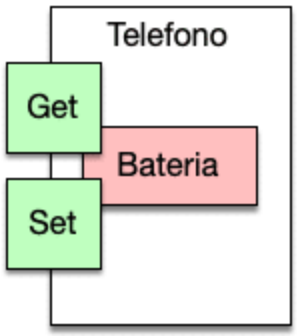
Debo aclarar algo en el caso del modificador de acceso protected y el acceso desde suclases. Es un error común pensar que se puede crear un objeto de la clase madre y luego acceder al atributo con acceso protected sin problemas, sin embargo esto no es cierto, puesto que el modificador protected lo que nos permite es acceder al atributo heredado desde el ámbito de la clase hija y no directamente. Sé que esto es un poco confuso así que te invito a ver el video explicativo que aparece al comienzo para quedar más claros.

Java Encapsulamiento y reutilización

¿Java Encapsulamiento ? . Cuando uno comienza a programar en Java el concepto de encapsulamiento es de los primeros que aparece y hace referencia a limitar el acceso a las variables de nuestras clases Java de tal forma que podamos tener un mayor control sobre ellas. Normalmente utilizando métodos set/get . La gente se queda contenta con esta respuesta y no le da más vueltas.

Eso si siempre le quedan a uno dudas de porque hay que usar el Eclipse para generar continuamente métodos set/get c**uando probablemente con variables publicas lo arreglaríamos todo de una forma mucho más directa .**

En programación muchas veces me he encontrado con que la “Fe” es una variable muy a tener en cuenta y que elimina análisis. Vamos a construir un ejemplo que nos ayude a entender porque el uso de la encapsulación nos puede ayudar a mejorar la reutilización de nuestro código y **su uso es más que recomendable.**



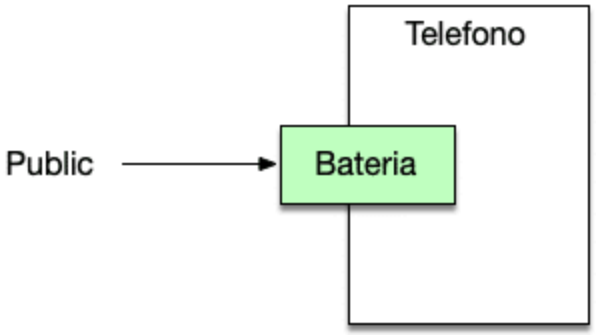
Supongamos que disponemos de una clase Java que se denomina Teléfono Esta clase es sencilla y solo tiene la marca y la capacidad de batería de nuestro Teléfono.

```
public class Telefono {
private String marca;
private int capacidad;
public String getMarca() {
return marca;
}
public void setMarca(String marca) {
this.marca = marca;
}
public int getCapacidad() {
```

```
return capacidad;
}
public void setCapacidad(int capacidad) {
this.capacidad = capacidad;
}
public Telefono(String marca, int capacidad) {
super();
this.marca = marca;
this.capacidad = capacidad;
}
public int duracionBateria() {
if (capacidad<3000) {
return 16;
}else {
return 24;
}
}
}
```

Disponemos de dos propiedades con sus métodos get/set y un constructor. ¿Realmente es necesario usar los métodos get/set? .

¿Sería suficiente con declarar los métodos públicos ? .Vamos a verlo



ejemplo2

```
public class Telefono {
public String marca;
public int capacidad;
public Telefono(String marca, int
capacidad){
super();
this.marca = marca;
this.capacidad = capacidad;
}
public int duracionBateria() {
if (capacidad < 3000) {
return 16;
} else {
return 24;
}
}
}
```

¿El porque de la encapsulación?

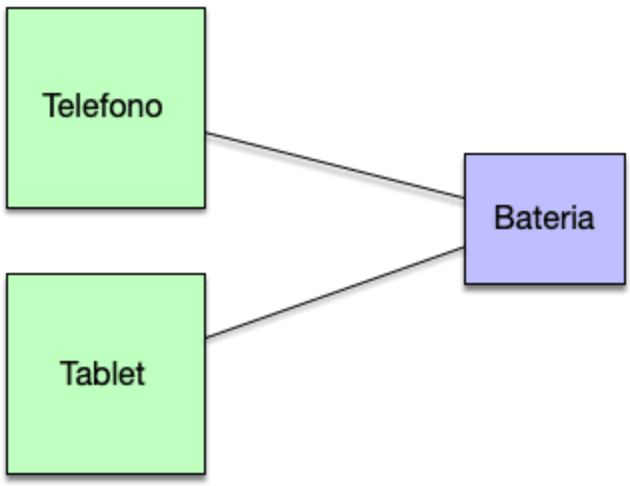
Es más que evidente que la clase queda simplificada y su funcionamiento es idéntico.
¿Entonces por qué tanta insistencia en usar los métodos get/set? . Imaginemonos que nosotros ahora disponemos de otra clase Tablet . Esta clase también dispondrá de una batería

```
public class Tablet {
private String marca;
private int capacidad;
public String getMarca() {
return marca;
}
public void setMarca(String marca) {
this.marca = marca;
}
public int getCapacidad() {
return capacidad;
}
public void setCapacidad(int capacidad) {
this.capacidad = capacidad;
}
public Tablet(String marca, int capacidad) {
super();
this.marca = marca;
this.capacidad = capacidad;
}
public int duracionBateria() {
if (capacidad<3000) {
return 16;
}else {
return 24;
}
}
}
```

Nos podemos dar cuenta que el código es muy similar y que nos vendría bien refactorizar el código de tal forma que existiera el concepto de batería.

```
package com.arquitecturajava.ejemplo3;
public class Bateria {
private int capacidad;
```

```
public int getCapacidad() {
return capacidad;
}
public void setCapacidad(int capacidad) {
this.capacidad = capacidad;
}
public Bateria(int capacidad) {
super();
this.capacidad = capacidad;
}
public int duracionBateria() {
if (capacidad < 3000) {
return 16;
} else {
return 24;
}
}
}
```



Java Delegación

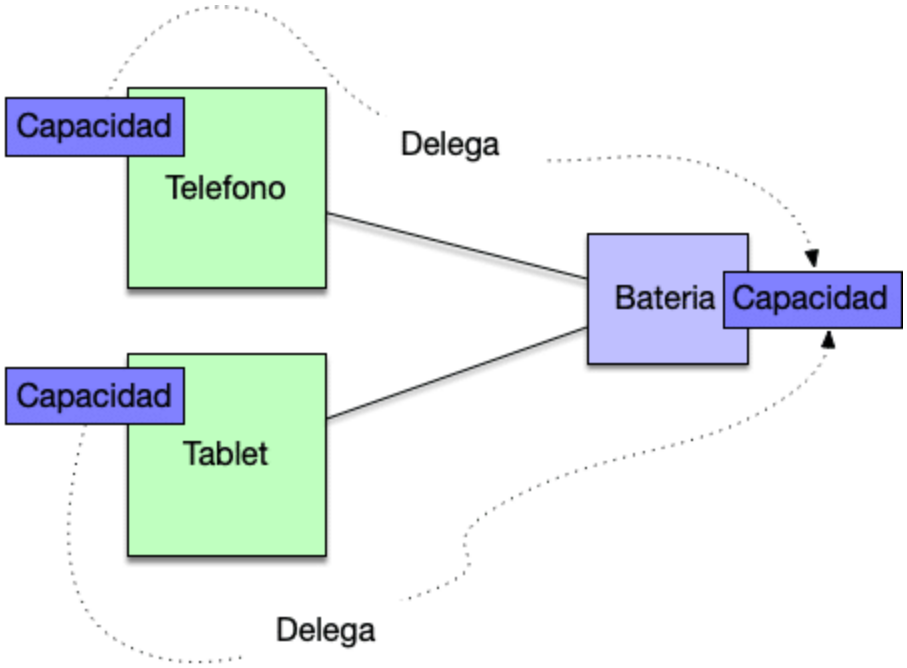
De esta forma podríamos reutilizar la bateria en nuestras clases utilizando el concepto de delegación (una clase delega en otra)

```
paejemplo3;
public class Telefono {
private String marca;
private Bateria bateria;
public String getMarca() {
return marca;
}
public void setMarca(String marca) {
this.marca = marca;
}
public int getCapacidad() {
return bateria.getCapacidad();
}
public void setCapacidad(int capacidad) {
bateria.setCapacidad(capacidad);
}
public Telefono(String marca, int capacidad) {
super();
this.marca = marca;
```

```
this.bateria= new Bateria(capacidad);
}
public int duracionBateria() {
return bateria.duracionBateria();
}
}

public class Tablet {
private String marca;
private Bateria bateria;
public String getMarca() {
return marca;
}
public void setMarca(String marca) {
this.marca = marca;
}
public int getCapacidad() {
return bateria.getCapacidad();
}
public void setCapacidad(int capacidad) {
bateria.setCapacidad(capacidad);
}
public Tablet(String marca, int capacidad) {
super();
this.marca = marca;
this.bateria= new Bateria(capacidad);
}
public int duracionBateria() {
return bateria.duracionBateria();
}
}
```

Estamos apoyándonos con el principio DRY al crear el concepto de batería y eso solo lo podemos conseguir usando el concepto de la encapsulación ya que nuestro nuevo código encapsula el concepto de batería mediante delegación.



El uso de la encapsulación nos permite construir estructuras más complejas relacionándolas e incrementando la flexibilidad y la capacidad de reutilización de nuestro código. Algo que el simple uso de las propiedades publicas no permite.