Tema 8: PAQUETES E INTERFACES

1. PAQUETES.

Los paquetes son contenedores de clases que se utilizan para mantener el espacio de nombres de clase dividido en compartimentos. Por ejemplo, un paquete permite que se cree una clase llamada **Caja**, que se puede almacenar en un paquete propio sin preocuparse por conflictos con otra clase llamada **Caja** almacenada en otro lugar. Los paquetes se almacenan de una manera jerárquica y se importan explícitamente en las definiciones de nuevas clases.

En general, un archivo fuente de Java puede contener las cuatro partes internas siguientes:
- ☐ Una única sentencia de paquete (opcional)
- ☐ Las sentencias de importación deseadas (opcional).
- ☐ Una única declaración de clase pública (obligatorio).
- ☐ Las clases privadas de paquete deseadas (opcional).

1.1. Comando package.

Hay que incluir el comando **package** como primera sentencia de un archivo fuente de Java. Cualquier clase que se defina dentro de ese archivo pertenece a ese paquete. La forma general de la sentencia **package**:

package nombrePaquete;

donde nombrePaquete es el nombre del paquete. Java utiliza los directorios del sistema de archivos para almacenar los paquetes.

Se puede crear una jerarquía de paquetes dentro de paquetes separando los niveles con puntos. La forma general de la sentencia package multinivel es la siguiente:

package nombrePaquete1[.nombrePaquete2[.nombrePaquete3..]];

1.2. Comando import.

Java incluye la sentencia **import** para que se puedan ver ciertas clases o paquetes enteros. Una vez importada, una clase puede ser referenciada directamente, utilizando solo su nombre. La sentencia **import** tiene como forma general:

import paquete1[.paquete2[.paquete3.(nombre_clase|*)]];

paquete1 es el nombre de un paquete de nivel superior, paquete2 es el nombre de un paquete subordinado contenido en el paquete exterior separado por un punto. NO hay

ningún límite práctico en la profundidad de la jerarquía de paquetes, solo el que imponga el sistema de archivos. Finalmente, se especifica un nombre_clase explícito o un asterisco (*), que indica que el compilador de Java debería importar ese paquete completo.
Todas las clases estándares de Java que se incluyen en la distribución de Java están almacenadas en un paquete llamado java. Las funciones del lenguaje básicas se almacenan dentro del paquete java llamado **java.lang**. Normalmente, se tiene que importar cada paquete o clase que se desea utilizar, pero debido a que Java no es útil sin gran parte de la funcionalidad de **java.lang**, el compilador la importa implícitamente para todos los programas. Es equivalente a tener al comienzo de todos los programas la siguiente línea:

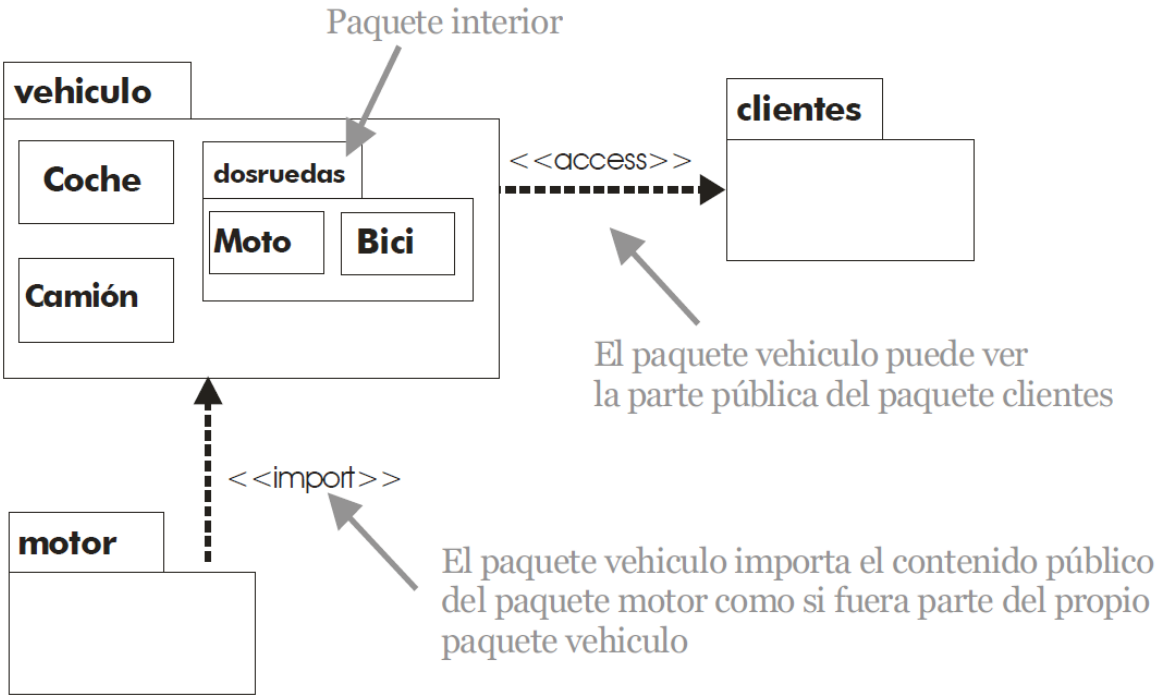
import java.lang.*;

Si existe una clase con el mismo nombre en dos paquetes diferentes que se importan utilizando el asterisco (por ejemplo), el compilador no dirá nada, a no ser que se intente utilizar una de las clases. En este caso, se produce un error de compilación y es necesario nombrar explícitamente la clase especificando su paquete.
Siempre que se utiliza el nombre de una clase, se puede incluir su nombre completamente cualificado, que incluye su jerarquía de paquetes completa. Este es un ejemplo de código que utiliza la sentencia **import**.

```
class MyDate extends java.util.Date{
}
```

1.3. Organización de los Paquetes.

Los paquetes en realidad son subdirectorios cuyo raíz debe ser absolutamente accesible por el sistema operativo. Para ello es necesario usar la variable de entorno **CLASSPATH** de la línea de comandos. Esta variable se suele definir en MI PC en el caso de Windows.
Hay que añadir a CLASSPATH las rutas a las carpetas que contienen los paquetes (normalmente todos los paquetes se suelen crear en la misma carpeta), a estas carpetas se las llama **filesystems.**
Así para el paquete **prueba.reloj** tiene que haber unacarpeta prueba, dentro de la cual habrá una carpeta reloj y esa carpeta prueba tiene que formar parte del **classpath.**
En los entornos de desarrollo o IDEs (NetBeans, JBuilder,...) se puede uno despreocupar de la variable classpath ya que poseen mecanismos de ayuda para gestionar los paquetes. Pero hay que tener en cuenta que si se compila a mano mediante el comando **java** se debe añadir el modificador**-cp** para que sean accesibles las clases contenidas en paquetes del classpath (por ejemplo **java -cp prueba.java).**
El uso de los paquetes permite que al compilar sólo se compile el código de la clase y de las clases importadas, en lugar de compilar todas las librerías. Sólo se compila lo que se utiliza.



2. INTERFACES.

A través de la palabra clave **interface**, Java permite abstraer totalmente la interfaz de su implementación. Utilizando **interface**, se puede especificar un conjunto de métodos que pueden ser implementados por una o más clases. La interfaz, en sí misma, no define realmente una implementación. Aunque son similares a las clases abstractas, las interfaces tienen una capacidad adicional; ya que una clase puede implementar más de un interfaz. Sin embargo, una clase puede heredar solo de una superclase (abstracta o no).

Utilizando la palabra clave **interface**, se puede especificar qué es lo que debe hacer una clase, pero no cómo lo hace. Las interfaces son sintácticamente como las clases, pero sin variables de instancia (objetos) y con métodos declarados sin cuerpo – considerados como estáticos -. Lo que esto significa en la práctica es que se pueden definir interfaces que no hacen suposiciones acerca de los detalles de implementación. Una vez definida una interfaz, cualquier número de clases puede implementarla. Además, una clase puede implementar cualquier número de interfaces.
Para implementar una interfaz, todo lo que necesita una clase es una implementación del conjunto completo de métodos de la interfaz. Sin embargo, cada clase es libre de determinar los detalles de su propia implementación. Con la palabra reservada **interface**, Java permite utilizar totalmente el aspecto del polimorfismo "una interfaz, múltiples métodos".
Las interfaces de Java están diseñadas para admitir resolución de método dinámica durante la ejecución. Para que un método de una clase sea llamado desde otra, ambas clases tienen que estar presentes durante la compilación, para que el compilador de Java pueda comprobar que el formato de los métodos es compatible. Este requisito provoca que el entorno de clases sea muy estático y no ampliable. Inevitablemente en un sistema como éste, la funcionalidad sube sistemáticamente en la jerarquía de clases, para que los mecanismos estén disponibles para más y más subclases. Las interfaces han sido diseñadas para evitar este problema al desconectar la definición de un método o conjunto de métodos de la jerarquía de herencias. Las interfaces están en una jerarquía distinta de la jerarquía de las clases, por lo que es posible que varias clases que no tengan la más mínima relación en cuanto a la jerarquía de clases implementen la misma interfaz. Aquí se destaca la potencia real de las interfaces.

Lo normal es que el nombre de las interfaces terminen con el texto "**able"** (*configurable*, *modificable, cargable*).
Por ejemplo en el caso de la clase Coche, esta deriva de la superclase Vehículo, pero además puesto que es un vehículo a motor, puede implementar métodos de una interfaz llamada por ejemplo **arrancable.** Se dirá entonces que la clase Coche es *arrancable*.

2.1. Definición de una Interface.

Una interfaz se define casi igual que una clase. La forma general de una interfaz es:

```
acceso interface nombre_interface{
     tipo_devuelto método1(lista_de_parámetros);
     tipo_devuelto método2(lista_de_parámetros);
     tipo_devuelto método3(lista_de_parámetros);
    //...
     tipo_devuelto métodoN(lista_de_parámetros);
     tipo var1=valor;
    //....
     tipo varN=valor;
}
```

Aquí acceso es **public** o no se utiliza. Cuando no se utiliza ningún modificador de acceso, entonces se aplica el acceso por defecto y la intefaz está solo disponible para otros miembros del paquete en el que ha sido declarada. Cuando se declara como **public**, la interfaz puede ser utilizada por cualquier código. Los métodos que se declaran no tienen cuerpo, terminando con un punto y coma después de la lista de parámetros. Son básicamente métodos abstractos ya que no puede haber implementación por defecto de un método especificado dentro de una interfaz. Cada clase que incluye una interfaz debe implementar todos los métodos.
Las variables se pueden declarar dentro de las declaraciones de la interfaz y son implícitamente **final** y **static**, lo que significa que no las puede cambiar la clase que las implementa, y además deben ser inicializadas con un valor constante. Todos los métodos y variables son implícitamente públicos si la interfaz ha sido declarada **public**.
Ejemplo,

```
interface CallBack{
  void callback(int param1);
}
```

2.2. Implementación de una Interface.

Una vez definida una interfaz, puede ser implementada por una o más clases. Para implementar una interfaz es necesario incluir la sentencia **implements** en la definición de la clase y crear los métodos definidos por la interfaz. La forma general de una clase que incluye la sentencia **implements** es:

```
acceso class nombre_de_clase [extends superclase] [implements interface1, interface2,...interfaceN] {
//cuerpo de la clase
}
```

Aquí, acceso es **public** o no se utiliza. Si una clase implementa más de una interfaz, las interfaces están separadas con una coma. Si una clase implementa dos interfaces que declaran el mismo método, entonces los clientes de cada interfaz utilizarán el mismo método. **Los métodos que implementan una interfaz deben ser declarados como**

public. Además, el formato del método implementado debe coincidir exactamente con el formato de tipo especificado en la definición de la interfaz.
Ejemplo,

```
class Ciente implements CallBack{
    //implementa la interface CallBack
    public void callback(int param)
    {
     System.out.println("callback llamado con : "+param);
    }
}
```

El método callback() ha sido declarado utilizando el modificador de acceso **public**.

Está permitido y es común que las clases que implementan interfaces definan otros miembros propios.
Por ejemplo,

```
class Ciente implements CallBack{
    //implementa la interface CallBack
    public void callback(int param)
    {
     System.out.println("callback llamado con : "+param);
    }

    void metodoPropioDeCliente(){
       System.out.println("Las clases que implementan interfaces pueden definir
    sus propios miembros: propiedades y métodos");
    }
}
```

2.3. Acceso a implementaciones a través de referencias de la interfaz

Se pueden declarar variables como referencias a objeto que utilizan una interfaz como tipo en lugar de una clase. Cualquier instancia de una clase que implementa la interfaz declarada se puede almacenar en una variable de ese tipo. Si se desea llamar a un método a través de una de estas variables, se llamará a la implementación correcta en base a la instancia de la interfaz que está siendo referenciada. El método al que llamar se determina dinámicamente durante la ejecución, lo que permite que se creen clases posteriores al código que llama a los métodos definidos en ellas.
Ejemplo,

```
class TestInter{
      public static void main(String args[]){
      CallBack c=new Cliente();
      c.callback(42);
      }
}
```

Salida: callback llamado con 42

Obsérvese que se ha declarado la variable c del tipo interfaz **Callback**, aunque se le ha asignado una instancia de **Cliente**. De esta manera, c solo se puede utilizar para acceder al método **callback()** y no a los otros miembros de la clase Cliente. Una variable referencia de una interfaz solo tiene conocimiento de los métodos que hay en la declaración de su interfaz, aunque c NO puede utilizar el método metodoPropioDelCliente() de la clase Cliente..

El siguiente ejemplo muestra el poder polimórfico de una referencia:

```
//Otra implementación de CallBack
class OtroCliente implements CallBack{

        //Implementa la interfaz CallBack

        public void callback(int p)
        {
                System.out.println("Otra version de callback");
                System.out.println("El cuadrado de p es :"+(p*p));
        }
}

class TestInter2{
        public static void main(String args[]){
        CallBack c=new Cliente();
        CallBack ob=new OtroCliente();

        c.callback(42);

        c=ob; // c hace referencia a un objeto OtroCliente
        c.callback(42);

        }
}
```

Salida:

```
callback llamado con 42
Otra versión de callback
El cuadrado de p es 1764
```

2.4. Implementación Parcial

Si una clase incluye una interface pero no implementa todos los métodos definidos por esa interfaz, entonces esa clase tiene que ser declarada como **abstract**.
Ejemplo,

```
abstract class Incompleta implements CallBack{
        int a, b;
        public void show(){
                System.out.println(a+ " "+b);
        }

        //NoImplementa el método callback de la interfaz CallBack

}
```

Cualquier clase que herede de la clase **Incompleta** tiene que implementar **callback()** o ser declarada como **abstract**.

2.5. Utilización de las interfaces

Ejemplo práctico de lo anteriormente expuesto:

```
public interface Apilable {
   public void push(int elemento); //añade un elemento de tipo entero
   public int pop();//elimina un elemento de una pila
}
```

La clase PilaFija implementa una pila de enteros de longitud fija.

```
public class PilaFija implements Apilable{
 private int pila[];
 private int numElementos;

   public PilaFija(int size) {
      this.pila=new int[size];
      this.numElementos=-1;
   }

   public void push(int elemento){
      if (this.numElementos == pila.length-1)
         System.out.println("La pila está llena");
      else
         this.pila[++this.numElementos]=elemento;
   }

 @Override
   public int pop(){
      if (this.numElementos <0)
      {
         System.out.println("La pila está vacía");
         return 0;
      }
      else
         return this.pila[this.numElementos--];
   }

}
```

La clase PilaDinamca implementa una pila de enteros de longitud variable.

```
public class PilaDinamica implements Apilable {
   private int pila[];
   private int numElementos;

   public PilaDinamica(int size) {
      this.pila=new int[size];
      this.numElementos=-1;
   }

  public void push(int elemento){
      if (this.numElementos == pila.length-1)
      {
         int temporal[]=new int[pila.length * 2]; //duplica el tamaño
         for (int i=0;i<pila.length;i++)
            temporal[i]=pila[i];
```

```
                pila=temporal;
                this.pila[++this.numElementos]=elemento;
            }
            else
                this.pila[++this.numElementos]=elemento;
        }

        @Override
        public int pop(){
            if (this.numElementos <0)
            {
                System.out.println("La pila está vacía");
                return 0;
            }
            else
                return this.pila[this.numElementos--];
        }
    }
```

```
public class TestInterfazPila {

    public static void main(String[] args) {
     PilaDinamica miPila1=new PilaDinamica(5);
     PilaDinamica miPila2=new PilaDinamica(8);

     for (int i=0;i<12;i++) miPila1.push(i);
     for (int i=0;i<20;i++) miPila2.push(i);

     System.out.println("Contenido de miPila1:");
     for (int i=0;i<12;i++) System.out.println(miPila1.pop());

     System.out.println("Contenido de miPila2:");
     for (int i=0;i<20;i++) System.out.println(miPila2.pop());

    }
}
```

La siguiente clase utiliza las implementaciones de PilaFija y PilaDinamica a través de una referencia de la interfaz. Esto significa que las llamadas a push() y pop() se resuelven en tiempo de ejecución y no durante la compilación.

```
public class Ppal {

    public static void main(String[] args) {
      Apilable apilable; //Crea una variable referencia de la interfaz

      PilaDinamica ds=new PilaDinamica(5);
      PilaFija fs=new PilaFija(8);

      apilable=ds; //apunta a la pila dinámica ds
      //almacena algunos números en la pila
      for(int i=0;i<12;i++) apilable.push(i);
```

```
        apilable=fs;
        for(int i=0;i<8;i++) apilable.push(i);

        apilable=ds;
        System.out.println("Valores de la pila dinámica ds ");
        for(int i=0;i<12;i++)
            System.out.println(apilable.pop());

        apilable=fs;
        System.out.println("Valores de la pila fija fs ");
        for(int i=0;i<8;i++)
            System.out.println(apilable.pop());
    }
}
```

En el anterior programa, apilable es una referencia a la interfaz Apilable. Así, cuando hace referencia a ds, utiliza las versiones de push() y pop() definidas en PilaDinamica. Cuando hace referencia a fs utiliza las versiones de push() y pop() definidas en PilaFija. El acceso a múltiples implementaciones de una interfaz a través de una variable referencia de la interfaz es la forma más poderosa que tiene Java para conseguir el polimorfismo en tiempo de ejecución.

2.5. Variables en Interfaces

Se pueden utilizar interfaces para importar constantes compartidas en múltiples clases simplemente declarando una interfaz que contenga variables que son inicializadas con los valores deseados. Cuando se incluye esta interfaz en una clase, es decir, cuando se implementa esa interfaz, todos esos nombres de variables estarán dentro del ámbito como **constantes**.

```
public interface Compartible {
    int NO=0;
    int SI=1;
    int QUIZAS=2;
    int MAS_TARDE =3;
    int PRONTO =4;
    int NUNCA=5;
}
```

```
import java.util.Random;
public class Cuestion implements Compartible{
    Random rand=new Random();

    public int pregunta(){
        int probabilidad=(int)(100*rand.nextDouble());
        if (probabilidad < 30) return NO;
        if (probabilidad < 60) return SI;
        if (probabilidad < 65) return QUIZAS;
        if (probabilidad < 75) return MAS_TARDE;
        if (probabilidad < 98) return PRONTO;
        return NUNCA;
    }
```

```
public class PreguntaMe implements Compartible {
   public static void respuesta(int resultado){
      switch(resultado){
         case NO: System.out.println("NO");break;
         case SI: System.out.println("SI");break;
         case QUIZAS: System.out.println("QUIZÁS");break;
         case MAS_TARDE: System.out.println("MÁS TARDE");break;
         case PRONTO: System.out.println("PRONTO");break;
         case NUNCA: System.out.println("NUNCA");
      }
   }
```

```
public class Principal {

   public static void main(String[] args) {
      Cuestion q=new Cuestion();

      PreguntaMe.respuesta(q.pregunta());
      PreguntaMe.respuesta(q.pregunta());
      PreguntaMe.respuesta(q.pregunta());
      PreguntaMe.respuesta(q.pregunta());
      PreguntaMe.respuesta(q.pregunta());
   }
}
```

Este programa utiliza la clase Random, que proporciona números aleatorios y contiene algunos métodos que permiten obtener eso números. El método que se utiliza es nextDouble(), que devuelve números aleatorios comprendidos entre 0.0 y 1.0.
Las dos clases: Cuestion y PreguntaMe, implementan la interfaz Compartible (constantes compartidas), donde se definen NO, SI, QUIZAS, MAS_TARDE, PRONTO y NUNCA. Dentro de cada clase, el código se refiere a estas constantes como si cada clase las hubiese definido o heredado directamente.

2.6. Las interfaces se pueden extender.

Una interfaz puede heredar otra utilizando la palabra clave extends. La sintaxis es la misma que se utiliza en la herencia de clases. Cuando una clase implementa una interfaz que hereda de otra, tiene que implementar todos los métodos definidos en la cadena de herencia de la interfaz.

E incluso, una clase puede implementar varias interfaces, con lo cual, es necesario implementar todos los métodos de todas las cadenas de herencia de cada una de las interfaces.

```
public interface A {
   public void metodo1();
   public void metodo2();
}
```

```
public interface B extends A{
   public void metodo3();
}
```

```
public class MiClase implements B{
   @Override
   public void metodo1(){
      System.out.println("Código de método 1");
   }
   @Override
   public void metodo2(){
      System.out.println("Código de método 2");
   }
   @Override
   public void metodo3(){
      System.out.println("Código de método 3");
   }
 }
```

```
public class Prueba {
   public static void main(String[] args) {
      MiClase ob=new MiClase();

      ob.metodo1();
      ob.metodo2();
      ob.metodo3();
   }
}
```

Si borramos la implementación de metodo1() en MiClase, se producirá un error de compilación.

2.7. Las interfaces pueden heredar más de una interface.

Además, una interfaz puede heredar de más de una interfaz, es decir, que en las interfaces, se puede producir la herencia múltiple.

Ejemplo,

```
public interface C {
   public void metodo4();
}
```

```
public interface B extends A,C {
   public void metodo3();
}
```

El interface B hereda los métodos: metodo1() y metodo2() de la interface A y metodo3() de la interface C.

Esto obliga a que la clase MiClase implemente también el método: metodo4(), que pertenece a la interface C.

```
public class MiClase implements B{
   @Override
   public void metodo1(){
      System.out.println("Código de método 1");
   }
   @Override
```

```
    public void metodo2(){
      System.out.println("Código de método 2");
    }
    @Override
    public void metodo3(){
      System.out.println("Código de método 3");
    }
    @Override
    public void metodo4(){
      System.out.println("Código de método 4");
    }
  }
```

Entonces en la clase Prueba (método main()), tenemos:

```
public class Prueba {
  public static void main(String[] args) {
    MiClase ob=new MiClase();

    ob.metodo1();
    ob.metodo2();
    ob.metodo3();
    ob.metodo4();
  }
}
```

2.8. Una clase puede implementar más de una interfaz

Por ejemplo,

```
public interface Prestable {
 public void prestar();
}
```

MiClase puede implementar las interfaces: B y Prestable.

```
public class MiClase implements B,Prestable{
  @Override
  public void metodo1(){ System.out.println("Código de método 1");  }
  @Override
  public void metodo2(){System.out.println("Código de método 2");}
  @Override
  public void metodo3(){System.out.println("Código de método 3");}
  @Override
  public void metodo4(){System.out.println("Código de método 4");}
  @Override
  public void prestar(){
    System.out.println("Metodo de la interfaz Prestable");
  }
}
```