TEMA 3

Programación Orientada a Objetos

La programación orientada a objetos es otra técnica para implementar el famoso dicho “divide y vencerás”. La idea es encapsular datos y métodos en objetos, de forma que cada objeto sea semiautónomo, encerrando métodos y datos privados o internos. De esta forma, el objeto puede interactuar con el resto del programa por medio de una interfaz bien definida por sus métodos públicos, es decir, invocando sus métodos desde fuera del objeto.

En POO un programa no es un código que llama a procedimientos, aquí un programa es un conjunto de objetos, independientes entre sí, que interaccionan entre ellos pasándose mensajes para llegar a resolver el problema en cuestión.

La POO fue creada para gestionar aplicaciones muy grandes y descomponerlas en unidades funcionales más pequeñas. En Java la POO gira en torno a unos conceptos clave como objeto, clase, miembro de dato, métodos y mensajes.

Conceptos Básicos de la Programación Orientada a Objetos

Objeto

Un objeto es un conjunto de variables y de los métodos relacionados con esas variables. Un objeto contiene en sí mismo la información y los métodos necesarios para manipular esa información.

Los objetos permiten tener un control total sobre ‘quién’ o ‘qué’ puede acceder a sus variables y métodos, es decir, pueden tener componentes públicos, a los que podrán acceder otros objetos, o componentes privados, a los que únicamente puede acceder él. Los componentes pueden ser tanto las variables como los métodos de ese objeto.

Un ejemplo de creación de objetos atendiendo a la definición “informal” dada en el objeto “coche” podría ser la siguiente:

```
coche c= new coche();
```

```
objeto coche
  variables internas
     color
     marca
  métodos del objeto
     damecolor
     asignamarca
```

Clase

Una clase es un prototipo o plantilla que define las variables y los métodos comunes a un cierto tipo de objetos. Son los moldes de los cuales se pueden crear múltiples objetos del mismo tipo. La clase define las variables y métodos comunes a los objetos del mismo tipo pero después, cada objeto tendrá sus propios valores para esas variables.

Lo primero es crear una clase antes de poder crear objetos o ejemplares de esa clase.

```
class coche {
     int color;
     String marca;
   public int damecolor {
        return color; }
   public String asignamarca {
        return marca}
  }
```

Ejemplo 3.1: Ejemplo de una clase.

Cuando se define una clase, se declara su forma y naturaleza exacta. Esto se hace especificando los datos que contiene la clase y el código que opera sobre ellos. Aunque las clases sencillas pueden contener sólo código o datos, la mayoría de las clases reales contienen las dos.

Miembros de una Clase

- **Variables**

Los miembros de datos definidos en una clase se llaman **variables de instancia**. Los datos compartidos por todos los objetos de una clase se denominan **variables de clase**. De acuerdo con estas definiciones, los miembros de una clase se pueden hacer accesibles desde fuera de la clase o bien que sean internos al objeto para su uso únicamente por los métodos del objeto.

Supongamos que tenemos un objeto definido de la clase coche, y que la variable color fuese de acceso público, es decir, una variable de clase:

Se define el objeto coche

```
coche c = new coche();
```

Y a partir del objeto creado se accedería a la variable color de la siguiente forma:

```
System.out.println(c.color);
```

De lo contrario, si la variable marca estuviese declarada como acceso no público únicamente se podrá acceder a su valor a través de los métodos definidos para tal fin en la clase que lo contiene.

- **Métodos**

La mayor parte de las veces es necesario utilizar métodos para acceder a las variables de instancia definidas en una clase. De hecho los métodos definen la interfaz de la mayoría de las clases. Esto permite que la clase oculte sus estructuras de datos internos detrás de un conjunto de métodos.

Los métodos se pueden dividir en los siguientes:

- Públicos, son aquellos que pueden ser accedidos desde fuera de la clase (***public***)
- Privados, son aquellos cuyo uso es interno a la clase (***private***)
- Protegidos, son usados únicamente por la misma clase y sus derivados (***protected***)

```
class clasepublica {
    int dato;

  public void escribedato {
    System.out.println(dato);
  }

}
```

```
class claseprivada {
    int dato;

  private int getdato {
    return dato;
  }
  public void escribedato {
    System.out.println(getdato());
  }

}
```

```
class claseprotegida {
    int dato;

  protected void escribedato {
    System.out.println(dato);
  }

}
```

Ejemplo 3.2: Ejemplo de tipos de acceso a métodos de una clase

- **Constructores**

Debido a que los requisitos de inicialización son tan comunes, Java permite que los objetos se inicialicen cuando son creados. Esta inicialización automática se realiza a través de un constructor.

Un constructor inicializa un objeto inmediatamente después de su creación. Tiene exactamente el mismo nombre de la clase en la que reside y sintácticamente es similar a un método. Una vez definido, se llama automáticamente al constructor después de crear el objeto, antes que se termine el operador "new". Los constructores no devuelven ningún tipo, ni siquiera void. Esto se debe a que el tipo implícito que devuelve un constructor de clase es el propio tipo de la clase. Es tarea del constructor inicializar todo el estado interno de un objeto para que el código que crea una instancia tenga un objeto íntegro y utilizable inmediatamente.

```
class Caja  {
        double ancho;
        double alto;
        double profundidad;

        //Constructor

        Caja()  {
                System.out.println("Constructor");
                ancho = 10;
                alto = 10;
                profundidad = 10;
        }
}
```

Ejemplo 3.3: Ejemplo de Constructor de una clase.

- **Creación de Objetos en Java**

Es importante saber que la declaración de una clase sólo crea un modelo o patrón, no crea un objeto real. En el ejemplo anterior no se crea ningún objeto del tipo Caja. Para crear un objeto es necesario primero declarar el objeto y luego crearlo a través del operador "new".

Para crear un objeto del tipo Caja es necesario utilizar las siguientes sentencias:

```
Caja miCaja;   //Declara un objeto tipo "Caja" llamado miCaja
miCaja = new Caja(); //Crea el objeto Caja llamado miCaja
```

Cuando se ejecute esta sentencia, miCaja será una instancia de Caja, es decir, tendrá realidad física.

Cada vez que se crea una instancia de una clase, se está creando un objeto que contiene su propia copia de las variables de instancia definidas por esa clase. Cada objeto del tipo Caja contendrá su propia copia de las variables de instancia ancho, alto y profundidad. Para acceder a estas variables se utiliza el operador '.'. Este operador relaciona el nombre del objeto con el nombre de una variable de instancia. Por ejemplo, para asignarle a la variable ancho de miCaja el valor 100 se utiliza la siguiente sentencia:

```
miCaja.ancho = 100;
```

Esta sentencia le dice al compilador que le asigne a la copia de la variable ancho que está contenida en el objeto miCaja el valor 100. En general, el operador punto se utiliza para acceder tanto a las variables de instancia como a los métodos de una clase.

Mensaje

Un mensaje es la invocación de un método de otro objeto. Un método es muy semejante a un procedimiento de la programación clásica. A un método se le pasan uno, varios o ningún dato y nos devuelve un dato a cambio.

Características de la POO

Encapsulamiento

El empaquetamiento de los datos y métodos se conoce como encapsulamiento. El objeto esconde sus datos de los demás objetos y permite el acceso a los datos mediante sus propios métodos. Esto recibe el nombre de ocultamiento de información. El encapsulamiento evita la corrupción de los datos de un objeto. Si todos los programas pudieran tener acceso a los datos de cualquier forma que quisieran los usuarios, los datos se podrían corromper o utilizar de manera incorrecta. El encapsulamiento protege los datos del uso arbitrario y no pretendido.

El encapsulamiento oculta los detalles de su implantación interna a los usuarios de un objeto. Los usuarios se dan cuenta de las operaciones que pueden solicitar del objeto, pero desconocen los detalles de cómo se lleva a cabo la operación. Todos los detalles específicos de los datos del objeto y la codificación de sus operaciones están fuera del alcance del usuario.

```
class Automovil  {
//Características o tipos de datos de la clase Automovil
        String estadoRuedas;
        boolean lucesEmergencia;
        double velocidad;

//Métodos o comportamientos de la clase Automovil
        Frenado()  {
                velocidad = 0;
                estadoRuedas = “Bloqueadas”;
                lucesEmergencia = true;
        }
}
```

Ejemplo 3.4: Ejemplo de Encapsulamiento de datos en una clase.

Herencia

La Herencia es el mecanismo por el cual se crean nuevos objetos definidos en términos de objetos ya existentes. Una subclase hereda propiedades de su clase padre, también llamada superclase. Una subclase también puede heredar propiedades de varias superclases. La subclase también puede tener sus propios métodos e incluso sus mismos tipos de datos.

Para heredar una clase, simplemente es necesario incorporar su definición en la definición de la otra clase utilizando la palabra clave “extends”.

```
class Caja  {
//Esta clase será superclase
        double ancho;
        double alto;
        double profundidad;
        Caja()  {
                System.out.println(“Constructor”);
                ancho = 10;
                alto = 10;
                profundidad = 10;
        }

        void volumen()   {
                System.out.println(“El volumen es: ”);
                System.out.println(ancho * alto * profundidad);
        }
 }
```

```
class CajaPesada extends Caja  {
//Esta clase será superclase
        double peso;
        CajaPesada()  {
                peso = 100;
        }

        public static void main(String args[])   {
                CajaPesada miCaja = new CajaPesada();
                miCaja.volumen();
                System.out.println(“Y el peso es: ” + miCaja.peso);
        }
 }
```

Ejemplo 3.5: Ejemplo de Herencia. Definición de la superclase Caja y la subclase CajaPesada.

En el ejemplo anterior, CajaPesada hereda todas las características de “Caja” y añade el componente peso. No es necesario que CajaPesada vuelva a crear todos los elementos de “Caja”, simplemente basta con extender de “Caja”.

- **Herencia Múltiple. Interface**

En Java no está permitida la herencia múltiple, es decir, no está permitido que una misma clase pueda heredar las propiedades y los métodos de varias clases padres. La explicación es que a la hora de crear este lenguaje de programación, se consideró que esta propiedad podría derivar en un aumento de la complejidad en el código. Sin embargo para no privar a Java de la potencia de la herencia múltiple, se introdujo un nuevo concepto de clase, la interface.

Una interface es igual que una clase con dos diferencias esenciales, sus métodos están vacíos, no hacen nada, y a la hora de definirla en vez de emplear la palabra clave “class” se emplea “interface”.

```
interface automovil {

        public int velocidad;
        public void acelera();
        public void frena();
        public void para();
}
```

Ejemplo 3.6: Declaración de la interface. Deben figurar todos los métodos que figurarán en la clase que la implemente.

La utilidad de la interface es la siguiente; cuando una clase implementa una interface lo que está haciendo es, digamos, una promesa de que esa clase va a implementar todos los métodos de esa interface. Si la clase no sobrescribe alguno de los métodos, esa clase se convierte en abstracta y no se podrá crear ningún objeto de ella.

Recordar que para que un método sobrescriba a otro ha de tener el mismo nombre, se le han de pasar los mismos datos, ha de devolver el mismo tipo de dato y ha de tener el mismo modificador de acceso (publica, privada o protegida como ya se verá más adelante).

Teniendo en cuenta la interface anterior veamos el siguiente ejemplo:

```
class coche1 implements  automovil  {
//Implementa todos los métodos contemplados en la interface automovil
        public int velocidad;

        coche1(){
                velocidad = 10;
        }
        public void acelera()  {
                velocidad++;
        }
        public void frena() {
                velocidad--;
        }
        public void para()  {
                velocidad = 0;
        }
}
```

Ejemplo 3.7: Ejemplo de una clase que implementa un Interface.

Polimorfismo

También llamado **sobrecarga de métodos**. En Java es posible definir dos o más métodos dentro de la misma clase que tengan el mismo nombre, pero con sus listas de parámetros distintas. Cuando ocurre esto, se dice que los métodos están sobrecargados y a este proceso se le denomina **sobrecarga de métodos**. La sobrecarga de métodos la utiliza Java para implementar el **polimorfismo**.

Cuando se invoca un método sobrecargado, Java utiliza el tipo y/o número de argumentos como guía para determinar la versión del método sobrecargado que realmente debe llamar. Por eso, los métodos sobrecargados deben diferenciarse en el tipo y/o número de sus parámetros.

```
/** Diversos modos de sumar */
public class Sumar {
  public float suma(float a, float b) {
    System.out.println("Estoy sumando reales");
    return a+b;
  }
  public int suma(int a, int b) {
    System.out.println("Estoy sumando enteros");
    return a+b;
  }
  public static void main(String[] args) {
    float x = 1, float y = 2;
    int v = 3, int w = 5;
    System.out.println(suma(x,y));
    System.out.println(suma(v,w));
  }
}
```

Ejemplo 3.8: Ejemplo de sobrecarga de métodos en Java (Polimorfismo)

Cuando Java encuentra una llamada a un método sobrecargado, simplemente ejecuta la versión del método cuyos parámetros coinciden con los argumentos utilizados en la llamada al método.

- **Sobrescritura de métodos**

En una jerarquía de clases, cuando un método de una subclase tiene el mismo nombre y tipo que un método de la superclase, entonces se dice que el método de la subclase sobrescribe al método de la superclase. Cuando se llama al método sobrescrito dentro de la subclase, siempre se refiere a la versión del método definida por la subclase. La versión definida por la superclase está oculta.

```
class Caja  {
        double ancho;
        double alto;
        double profundidad;

        Caja()  {
                System.out.println("Constructor");
                ancho = 10;
                alto = 10;
                profundidad = 10;
        }
        void volumen()   {
                System.out.println("El volumen es: ");
                System.out.println(ancho * alto * profundidad);
        }
        void volumen(double w, double, h, double p)   {
                ancho = w;
                alto = h;
                profundidad = p;
                System.out.println("El volumen es: ");
                System.out.println(ancho * alto * profundidad);
        }
        public static void main(String args[])  {
                Caja miCaja = new Caja();
                miCaja.volumen();
                miCaja.volumen(10,15,20);
        }
 }
```

Ejemplo 3.9: Otro ejemplo de Polimorfismo. Sobrecarga de métodos.

- **Sobrecarga de Constructores**

Además de la sobrecarga de métodos normales, también se pueden sobrecargar los constructores. de hecho, en la mayoría de las clases que se implementan en el mundo real, la sobrecarga de constructores es la norma y no la excepción.

```
class Caja  {
        double ancho;
        double alto;
        double profundidad;

        Caja()  {
                System.out.println("Constructor");
                ancho = 10;
                alto = 10;
                profundidad = 10;
        }

        Caja(double w, double h, double p)  {
                System.out.println("Constructor con tres parámetros");
                ancho = w;
                alto = h;
                profundidad = p;
        }


        void volumen()   {
                System.out.println("El volumen es: ");
                System.out.println(ancho * alto * profundidad);
        }

        void volumen(double w, double, h, double p)   {
                ancho = w;
                alto = h;
                profundidad = p;
                System.out.println("El volumen es: ");
                System.out.println(ancho * alto * profundidad);
        }

        public static void main(String args[])  {
                Caja miCaja1 = new Caja(4,4,4);
                miCaja1.volumen();
                Caja miCaja2 = new Caja();
                miCaja2.volumen();
                miCaja2.volumen(9,9,9);
        }
 }
```

Ejemplo 3.10: Ejemplo de Polimorfismo. Sobrecarga de constructores.

- **Sobrescritura de métodos**

En una jerarquía de clases, cuando un método de una subclase tiene el mismo nombre y tipo que un método en la superclase, entonces se dice que **el método de la subclase sobrescribe al método de la superclase**. Cuando se llama al método sobrescrito dentro de la subclase, siempre se refiere a la versión del método definida por la subclase. La versión del método definida por la superclase está oculta.

```
class CajaPesada extends Caja  {

        double peso;
        CajaPesada()  {
                ancho = 10;
                alto = 10;
                profundidad = 10;
                peso = 100;
        }

        void volumen()  {
                System.out.println("El volumen es: " + ancho * alto * profundidad);
                System.out.println("El peso es: " + peso);
        }

        public static void main(String args[])  {
                CajaPesada miCaja = new CajaPesada();
                miCaja.volumen();
        }
}
```

Ejemplo 3.11: Sobrescritura de métodos.
**** Comentar el método volumen y ejecutar de nuevo ****

Las palabras claves “this” y “super”

this

Algunas veces un método necesita hacer referencia al objeto que lo invocó. Para permitir esto. Java define la palabra clave “this”, que puede ser utilizada dentro de cualquier método para referirse al objeto actual, “this” es siempre una referencia al objeto sobre el que ha sido llamado el método. Se puede utilizar “this” siempre que se quiera una referencia a un objeto del tipo de la clase actual.

```
void volumen(double ancho, double, alto, double profundidad)   {
        this.ancho = ancho;
        this.alto = alto;
        this.profundidad = profundidad;
}
```

Ejemplo 3.12: Ejemplo this.

super

Se puede utilizar de dos formas. La primera para llamar al constructor de la superclase. La segunda se utiliza para acceder a un miembro de la superclase que ha sido ocultado por un miembro de la subclase.

```
class CajaPesada extends Caja  {

        double peso;
        CajaPesada()  {
                super(20,20,20);
        }

        void volumen()   {
                System.out.println(“El volumen es: ” + ancho * alto * profundidad);
                System.out.println(“El peso es: ” + peso);
        }

        public static void main(String args[])   {
                CajaPesada miCaja = new CajaPesada();
                miCaja.volumen();
        }
 }
```

Ejemplo 3.13: Ejemplo super.

Variables y métodos estáticos

Hay ocasiones en las que se necesita definir un miembro de una clase que será utilizado independientemente de cualquier objeto de esa clase. Normalmente a un miembro de una clase se debe acceder a través de un objeto de esa clase. Sin embargo, es posible crear un miembro que pueda ser utilizado por sí mismo sin referirse a una instancia específica.

Para crear un miembro de ese tipo es necesario preceder su declaración con la palabra clave "**static**".

Cuando se declara un miembro como "**static**", se puede acceder a él antes de que se haya creado ningún objeto de esa clase, y sin hacer referencia a ningún objeto. Se pueden declarar "**static**" tanto los métodos como las variables.

Las variables de instancia declaradas como "**static**" son, básicamente, variables "globales". Cuando se declaran objetos de su clase, no se hace ninguna copia de las variables estáticas. De hecho, todas las instancias de la clase comparten la misma variable estática.

Los métodos declarados estáticos tienen una serie de restricciones:

- Sólo pueden llamar a otros métodos estáticos ("static")
- Sólo pueden acceder a datos estáticos
- No se puede referir a "this"

Los siguientes ejemplos muestran el uso de la palabra clave "static" para declarar variables y métodos estáticos

```
class EjemploStatic  {

        static int a = 20;
        static int b = 12;
        static void imprime()
          {
                System.out.println("a = " + a);
          }

}

class UtilizaEjemploStatic  {

        public static void main(String args[])
          {
                System.out.println("b = " + EjemploStatic.b);
                EjemplolStatic.imprime();
          }

}
```

Ejemplo 3.14: Utilización variables y métodos static.

Modificadores de acceso

Java proporciona muchos niveles de protección para permitir un control preciso de la visibilidad de las variables y métodos. Las clases y los paquetes son los medios de encapsular y contener el espacio de nombres y el ámbito de las variables y métodos. Java proporciona unos cuantos mecanismos más para permitir un control de acceso incluso más preciso en circunstancias diferentes.

Es obvio que dentro de una clase, todas las variables y métodos son visibles para todas las otras partes de la misma clase, dado que la clase es la unidad de abstracción más pequeña en Java. Por la existencia de paquetes, Java debe distinguir cuatro categorías de visibilidad entre elementos de clase:

- Subclases del mismo paquete
- Clases diferentes en el mismo paquete
- Subclases en paquetes distintos
- Clases que no están ni en el mismo paquete ni en las subclases

Las tres palabras clave, "**private**", "**public**" y "**protected**" se combinan de varias maneras para generar muchos niveles de acceso. La siguiente tabla resume las iteraciones. Las columnas muestran las combinaciones legales de los tres modificadores de protección, mientras que las filas indican el lugar desde el cual se quiere acceder. El valor de cada celda dice si tendrá éxito un acceso desde un sitio en particular (la fila) a una variable declarada con un cierto acceso (la columna).

	Sin modificador	**private**	**protected**	**public**
Misma clase	SI	SI	SI	SI
Otra clase del mismo paquete	SI	NO	SI	SI
Clase de diferente paquete	NO	NO	NO	SI
Subclase del mismo paquete	SI	NO	SI	SI
Subclase de diferente paquete	NO	NO	SI	SI

Clases Abstractas

Cuando se desarrolla una jerarquía de clases, puede producirse que algún comportamiento está presente en todas ellas pero se materializa de forma distinta. Un ejemplo podría ser una estructura de clases para manipular figuras geométricas. Se tendría una clase genérica, que podría llamarse FiguraGeometrica y una serie de clases que extienden a la anterior que podrían ser Circulo, Poligono, etc.

Podría haber un método dibujar común a todas las clases anteriores, pero esta operación se llevaría a cabo de forma diferente en el caso de cada figura. Por otra parte la acción dibujar no tiene sentido para la clase genérica FiguraGeometrica, ya que esta clase representa una abstracción del conjunto de figuras posibles.

Para resolver esta problemática Java proporciona las clases y métodos abstractos. Un **método abstracto** es un método declarado en una clase para el cual esa clase no proporciona la implementación (el código). Una clase abstracta es una clase que tiene al menos un método abstracto. Una clase que extiende a una clase abstracta debe implementar los métodos abstractos (escribir el código) o bien volverlos a declarar como abstractos, con lo que ella misma se convierte también en clase abstracta.

```
abstract class FiguraGeometrica {
   . . .
   abstract void dibujar();
   . . .
}
class Circulo extends FiguraGeometrica {
   . . .
   void dibujar() {
      // codigo para dibujar Circulo
      . . .
   }
}
```

Ejemplo 3.15: Declaración de una clase abstracta y otra que extiende de ella.

Imaginemos que se crea una clase llamada 'a' que tiene un método print, que visualiza una cadena de caracteres obteniéndola de una llamada al método getData:

```
class a {
   String getData();
   public void print()  {
         System.out.println(
getData());

   }
}
```

El método getData no tiene ninguna implementación en la clase 'a' ya que se quiere que los desarrolladores especifiquen qué datos se van a visualizar. Para hacer que éstos sepan que deben implementar el método getData, se puede hacer abstracto, lo que significa que también la clase lo será. Por lo tanto la definición de la clase 'a' quedaría de la siguiente manera:

```
abstract class a {

   abstract String getData();

   public void print()  {
      System.out.println(getData());
   }
}
```

Cuando se crea una subclase de a, hay que proporcionar una implementación de getData, como es la siguiente:

```
class b extends a{

   String getData()  {
     return “Hola desde Java!!”;
   }
}
```

En la clase principal de la aplicación sería así la llamada al método print de este objeto:

```
public class app{

  public static void main(String[] arg) {

      b b1 = new b();
      b1.print();
   }
}
```

Ejemplo 3.16: Llamada de un método de una clase abstracta

Paquetes en Java

Los paquetes son una forma de organizar grupos de clases. Un paquete contiene un conjunto de clases relacionadas bien por finalidad, por ámbito o por herencia.

Los paquetes resuelven el problema del conflicto entre los nombres de las clases. Al crecer el número de clases crece la probabilidad de designar con el mismo nombre a dos clases diferentes.

- **La sentencia “import”**

Para importar clases de un paquete se usa el comando “import”. Se puede importar una sola clase:

```
import java.awt.Font;
```

O bien se pueden importar las clases declaradas públicas de un paquete completo, utilizando para ello, un asterisco para reemplazar los nombres de clase individuales:

```
import java.awt.*;
```

Para la creación de un objeto de la clase Font podemos seguir elegir dos formas:

```
import java.awt.*;

Font fuente = new Font(“Monospaced”, Fold.BOLD, 36);
```

o bien sin la sentencia import con lo que se debería escribir la localización de la clase que queremos utilizar para la creación del objeto:

```
java.awt.Font fuente = new java.awt.Font("Monospaced", Font.BOLD, 36);
```

- **Principales paquetes en Java**

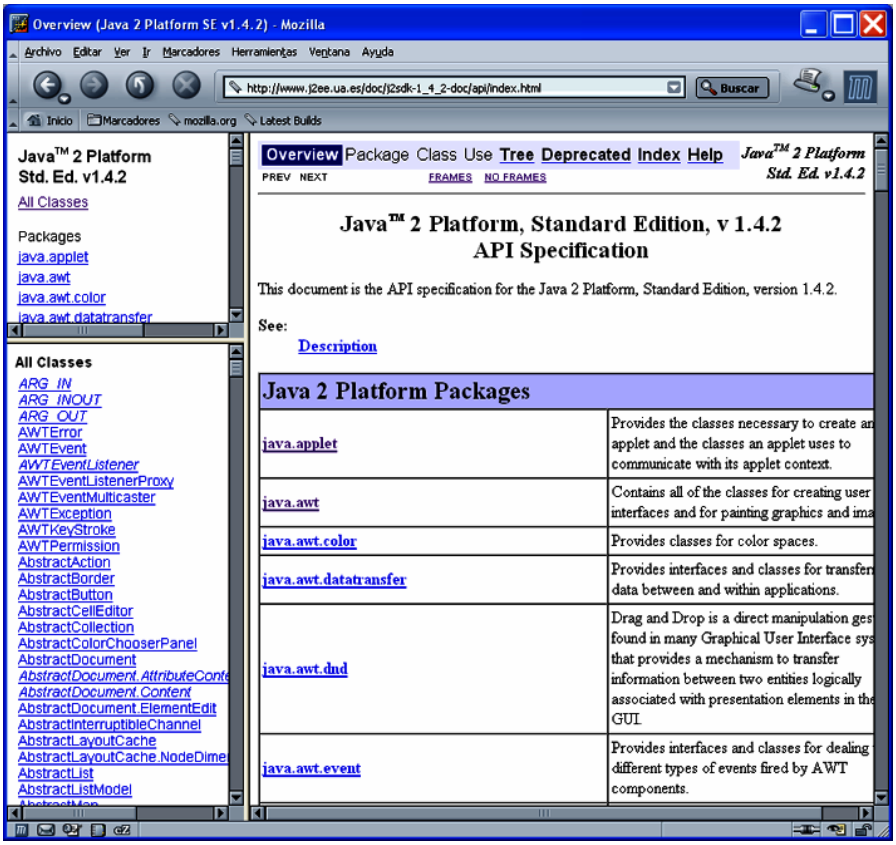
Java proporciona una serie de paquetes que incluyen ventanas, utilidades, sistema de entrada/salida general, herramientas de comunicaciones, etc... En la versión del JDK, los paquetes Java que se incluyen son los siguientes:

- **java.applet** Este paquete contiene clases diseñadas para usar con applets. Hay una clase Applet y tres interfaces: AppletContext, AppletStub y AudioClip.
- **java.awt** El paquete Abstract Windowing Toolkit (awt) contiene clases para generar widgets y componentes GUI (Interfaz Gráfico de Usuario). Incluye las clases Button, Checkbox, Choice, Component, Graphics, Menu, Panel, TextArea y TextField.
- **java.io** El paquete de entrada/salida contiene las clases de acceso a ficheros: FileInputStream y FileOutputStream.

- **java.lang** Este paquete incluye las clases del lenguaje Java propiamente dicho: Object, Thread, Exception, System, Integer, Float, Math, String, etc.
- **java.net** Este paquete da soporte a las conexiones del protocolo TCP/IP y, además, incluye las clases Socket, URL y URLConnection.
- **java.util** Este paquete es una miscelánea de clases útiles para muchas cosas en programación. Se incluyen, entre otras, Date (fecha), Dictionary (diccionario), Random (números aleatorios) y Stack (pila FIFO).

Los principales paquetes que componen Java se pueden y deben consultar en

http://java.sun.com/j2se/1.4.2/docs/api/index.html



LABORATORIO

A continuación se muestra el código del archivo DistanciaPunto.java, el cual contiene dos clases; Punto2D y DistanciaPunto.

La clase Punto2D es una representación de tipo de dato Punto en dos dimensiones, para el cual se definen dos variables x e y, las cuales representan las coordenadas de un punto en el plano. La clase Punto2D también contiene un método “distancia”, por medio del cual se puede calcular la distancia euclidea del objeto Punto2D desde el cual se llama al método, con respecto a otro objeto Punto2D o con respecto a un par (x,y), que representa un punto en el plano.

La clase DistanciPunto se utiliza para crear objetos del tipo Punto2D. En este ejemplo se utiliza el método estático “sqrt” de la clase Math para calcular la raíz cuadrada de su parámetro.

```
class Punto2D
  {
        int x,y;
        double valor;
        Punto2D(int x, int y)
          {
                this.x = x;
                this.y = y;
          }
        double distancia(int x, int y)
          {
                int dx = this.x – x ;
                int dy = this.y – y;
                valor = dx * dx + dy * dy;
                return Math.sqrt(valor);
          }
        double distancia(Punto2D p)
          {
                return distancia(p.x , p.y) ;
          }
}
```

```
public class DistanciaPunto
  {
    public static void main(String args[])
      {
        Punto2D p1 = new Punto2D(0,0);
        Punto2D p2 = new Punto2D(30,40);
        prt(“p1 = ” + p1.x + “,” + p1.y) ;
        prt(“p2 = ” + p2.x + “,” + p2.y) ;
        prt(“p1.distancia(p2) ” +
             p1.distancia(p2)) ;
        prt(“p2.distancia(p1) ” +
             p2.distancia(p1)) ;

      }

    public void prt(String s)
      {
        System.out.println(s);
      }
}
```

Observa el siguiente fragmento de código y comprueba si funciona

```
class alfa {

    private int j;
    private int k;

    void sumarAj(int operando)
      {j+=operando;}
}

class beta {

  public static void main(String args[])
    {
        alfa obAlfa = new alfa();
        obAlfa.sumarAj(2);
        obAlfa.k = 4;
    }
}
```

Acertijo del Pedro ladrador.

Fíjese bien el la sencilla jerarquía de clases siguiente:

```
class Perro {

    public static void ladrido() {
       System.out.println("Guau");
    }
}

class Basenji extends Perro {
    public static void ladrido()  {}
}

public class Ladrido

{
    public static void main(String args[])
     {
        Perro ladrador = new Perro();
        Perro mordedor = new Basenji();
        ladrador.ladrido();
        mordedor.ladrido();
     }
}
```

La pregunta es sencilla, cuántas veces “ladra” el programa??

Juego de Palabras Tonto

Este programa es bastante obvio, pero.... qué imprime?

```
public class Trivial {

        public static void main(String[] args)  {
                System.out.print(“H” + “a”);
                System.out.print(‘H’ + ‘a’);

        }
}
```

Repasando Herencias

Cree una subclase de CajaPesada que herede todas las características de Caja y CajaPesada, y añada un campo costo, el cual debe guardar el costo del envío de un paquete.

EJERCICIOS

Cree una clase NumeroDecimal, defina dos propiedades (variables o atributos) que son: entero, decimal; como lo indican sus nombres, representan a un número decimal.

Cree los métodos para la clase NumeroComplejo así:

- Método que muestre en pantalla el valor de cada propiedad
- Método que inicialice el objeto con cualquier valor (constructor)
- Método que reciba un número decimal y devuelva el cuadrado del mismo
- Método que calcule el factorial de la parte entera de este objeto.

Implemente la anterior clase con una clase principal y ejecute los métodos anteriores.

Cree la clase abstracta FIGURAS2D que permita DEFINIR los métodos CalculaArea y CalculaPerimetro. Además deberá crear tres subclases de FIGURAS que implementen o desarrollen dichos métodos (Cuadrado,Círculo,Triángulo)

Area = Lado * Lado
Perimetro = 2 * (Lado + Lado)

Cree la clase abstracta NUMEROS que tenga dos atributos: NUM1 y NUM2, en la cual estarán definidos los métodos SUMA y RESTA. Además cree una subclase de NUMEROS que implemente sus métodos con la particularidad de que esta subclase sólo podrá operar con números del 0 al 9