**PRIMER PARCIAL - PROGRAMACION AVANZADA 1**

**La clase Object**

Todas las clases son en realidad subclases de una clase más amplia: la clase Object. Cuando decimos esto estamos diciendo que todas las clases que utilicemos hasta las que creamos se extienden de la clase Object.

* **public boolean equals(Object obj):** compara si dos objetos son iguales, por defecto un objeto es igual solamente a sí mismo
* **public int hashCode ():** Devuelve un valor de código hash para el objeto. Este método se apoya en beneficio de tablas hash tales como los proporcionados por java.util.Hashtable
* **protected Object clone() throws CloneNotSuportedException**: devuelve una copia binaria del objeto, al hacer la copia hace referencia a una nueva posición de memoria.
* **public final Class getClass():** devuelve el objeto del tipo Class que representa dicha clase durante la ejecución, es decir devuelve el tipo de clase al que pertenece.
* **protected void finalize() throws Throwable:** se usa para finalizar el objeto, es decir, se avisa al administrador de la memoria que ya no se usa dicho objeto, y se puede ejecutar código especial antes de que se libere la memoria.
* **public String toString():** devuelvo una cadena describiendo el objeto.
* **void nativa public final notify():** Se despierta un solo hilo que está esperando en el monitor de este objeto. Un subproceso espera en el monitor de un objeto llamando a uno de los de espera métodos
* **final public native void notifyAll ():** Se despierta todos los temas que están en espera en el monitor de este objeto. Un subproceso espera en el monitor de un objeto llamando a uno de los de espera métodos.
* **final public void wait():** Espera que se le notifique por otro subproceso de un cambio en este objeto.

Las clases derivadas deben sobrescribir los métodos adecuadamente. También habría que utilizar la implementación de la Interface Cloneable, en el caso que queramos clonar el Objeto.

**La clase String**

La clase String está orientada a manejar cadenas de caracteres y proporciona métodos para el tratamiento de las cadenas de caracteres: acceso a caracteres individuales, buscar y extraer una subcadena, copiar cadenas, convertir cadenas a mayúsculas o minúsculas, etc.

Debemos saber que los objetos String no son modificables. Por lo tanto, los métodos que actúan sobre un String con la intención de modificarlo lo que hacen es crear un nuevo String a partir del original y devolverlo modificado.

**Documentación:**

/\*\* \* Trae un Beneficio por número, año y abreviatura de plan que

\*no esté en estado BAJA   
\*

\* @param año: año del beneficiario

\* @param numero: número del beneficiario

\* @Param abreviatura: abreviatura del plan

\* @return BeneficioBean con todos sus datos cargados

\* @throws AccesoDatosException: Error al conectarse a la base de datos \*/

@author Nombre del desarrollador. Nombre autor o autores

@deprecated Indica que el método o clase es obsoleto (propio de versiones anteriores) y que no se recomienda su uso.

@see Asocia con otro método o clase.

@version Versión del método o clase

**Excepciones:**

Las excepciones en Java son objetos. Todos los tipos de excepciones deben extender de la clase Throwable o una de sus subclases. Por convención, los nuevos tipos de excepciones extienden de la subclase Exception.

Las excepciones son lanzadas usando la cláusula throw

La primer cosa que el método de arriba hace es declarar cuales excepciones chequeadas serán lanzadas. Java requiere la declaración de excepciones chequeadas que el método lance, debido a que los programadores que la utilicen deben saber las posibles excepciones que lancen el método así como el tipo de valor que retorne.

Las excepciones son lanzadas por uno de estos posibles escenarios:

• Se llama un método que lanza una excepción, por ejemplo, el método readline () de DataInputStream.

• Se detecta un error y lanza una excepción con la sentencia throw

• Se produce un error de programa. • Un error interno ocurre en Java.

**Finally**: Se utiliza para limpiar el estado interno o para liberar recursos que no son de objetos.

**Try con recursos**: A partir de java 7 se incluyeron los try con recursos, para solucionar el inconveniente de tener que cerrar archivos o conexiones a base de datos en el finally.

**Test Unitario**

Una prueba unitaria, en programación, es una forma de probar el correcto funcionamiento de un módulo de código. Esto sirve para asegurar que cada uno de los módulos funcione correctamente por separado. Luego, con las Pruebas de Integración, se podrá asegurar el correcto funcionamiento del sistema o subsistema en cuestión.

**Características**

• **Automatizable**: No debería requerirse una intervención manual, los test unitarios deben ejecutarse de forma automática.

• **Completas**: Deben cubrir la mayor cantidad de código. Para que estos nos rindan al 100% debemos intentar testear todo nuestro código unitariamente.

• **Repetibles o Reutilizables**: No se deben crear pruebas que sólo puedan ser ejecutadas una sola vez. Porque lo que queremos es testear nuestra aplicación de forma automática cada vez que realizamos un cambio.

• **Independientes**: La ejecución de una prueba no debe afectar a la ejecución de otra, ya que si un test falla, el resto podrían fallar debido a que estaban conectados entre sí.

• **Profesionales**: Las pruebas deben ser consideradas igual que el código, con la misma profesionalidad, documentación, etc. Nosotros tenemos que programar y testear de forma tal que cualquier persona pueda entender que es lo que estamos haciendo y porque.

**Ventajas**

Una de las principales ventajas que tenemos es fomentar el cambio y el orden. Otra de las ventajas que nos brinda es simplificar las implementaciones.

**JUnit**

Es un framework, como bien dije antes, que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera

**Funciona de la siguiente manera:** En función de algún valor de entrada se evalúa el valor de retorno esperado. Si la clase cumple con la especificación, entonces JUnit devolverá que el método de la clase pasó exitosamente la prueba. En caso de que el valor esperado sea diferente al que regresó el método durante la ejecución, JUnit devolverá un fallo en el método correspondiente.

**File:**

La clase File se puede utilizar para operaciones sobre directorios y archivos:

• Si un archivo o directorio existe se utiliza el método **exists**()

• Saber si un objeto File es archivo o directorio se utilizar el método **isFile** () o **isDirectory** ()

• En caso de directorio, si no existe crearlo. El método **mkdir** () crea un directorio y **mkdirs** () crea la ruta completa.

• El método **list**() lista de todos los nombres de archivos del directorio.

• El método **listFiles**() lista de todos los archivos del directorio.

**Colecciones**

Las colecciones son una especie de arrays de tamaño dinámico. Para usarlas haremos uso del Java Collections Framework (JCF), el cual contiene un conjunto de clases e interfaces del paquete java.util para gestionar colecciones de objetos. En Java las principales interfaces que disponemos para trabajar con colecciones son: Collection, Set, List, Queue y Map.

**Patrones de diseño**

Hay diferentes tipos de patrones de diseño, se dividen en:

**Patrones Creacionales**: tratan con las formas de crear instancias de objetos. El objetivo de estos patrones es de abstraer el proceso de instanciación y ocultar los detalles de cómo los objetos son creados o inicializados.

**Patrones Estructurales**: describen como las clases y objetos pueden ser combinados para formar grandes estructuras y proporcionar nuevas funcionalidades. Estos objetos adicionados pueden ser incluso objetos simples u objetos compuestos.

**Patrones de Comportamiento**: nos ayudan a definir la comunicación e iteración entre los objetos de un sistema. El propósito de este patrón es reducir el acoplamiento entre los objetos

**Creacionales**: Factory, Singleton.

**Estructurales:** Adapter

**Factory**

Propósito: Crear distintos objetos de la misma naturaleza en tiempo de ejecución

**Singleton**

Propósito: Asegurar que una clase sólo tiene un ejemplar, y proporcionar un punto de acceso global a éste

**Facade**

Propósito: Simplifica el acceso a un conjunto de objetos proporcionando uno que todos los clientes pueden usar para comunicarse con el conjunto

**Adapter**

Propósito: Convierte la interfaz de una clase en otra interfaz que el cliente

**State**

Propósito: Permite a un objeto alterar su comportamiento según el estado interno en que se encuentre

**Hilos:**

Un hilo es un proceso que se está ejecutando en un momento determinado en nuestro sistema operativo, como cualquier otra tarea, esto se realiza directamente en el procesador

**Equals y hashCode**

@Override

    public boolean equals(Object obj) {

        if(obj != null) {

            //*Comparar Clases*

            if( obj.getClass().equals(this.getClass())) {

                MiClase mC = (MiClase)obj;

                if(mC.getTexto().equals(this.getTexto()) && mC.numero == this.numero ) {

                    return true;

                }

            }

        }

        return false;

    }

    @Override

    public int hashCode() {

        int primo = 31;

        int resultado = 1;

        resultado = this.numero \* primo;

        resultado += primo \* this.getTexto().hashCode();

        return resultado;

    }

**Para obtener por teclado**

Scanner sc = new Scanner(System.in);

miCalculadora.numero1 = Double.parseDouble(sc.nextLine());

Assert.assertTrue(true);

Assert.fail();

Assert.assertEquals(0, resultado);

public class Ingreso implements IEstado {

    static Ingreso i;

    private Ingreso() { }

    public static Ingreso getInstance() {

        if(i == null) {

            i = new Ingreso();

        }

        return i;

    }

    @Override

    public String obtenerEstado(Alumno a) {

        return "Ingreso";

    }

    @Override

    public void cambiarEstado(Alumno a, int cambiar) {

        a.setEstado(Regular.getInstance());

    }

}

PARA LEER ARCHIVO

try(BufferedReader br = new BufferedReader(

            new FileReader("D:\\algo.txt")

        )) {

            String linea;

            while ((linea = br.readLine()) != null) {

                System.out.println(linea);

            }

        } catch (IOException e) {

            e.printStackTrace();

        }

PARA ESCRIBIR EN EL ARCHIVO

File f = new File("D:\\","algo.txt");

        try {

            if(f.exists()) {

                System.out.println("Existe");

                //*Para escribir en el archivo*

                //*true sobreescribe*

                FileWriter fw = new FileWriter(f,false);

                //*De que manera escribira el archivo*

                BufferedWriter bw = new BufferedWriter(fw);

                bw.write("Texto");

                //*Limpia el buffer y escribe el disco*

                bw.flush();

                //*Cierra el archivo*

                bw.close();

            } else {

                f.createNewFile();

                System.out.println("Archivo creado");

            }

        } catch (IOException e) {

            e.printStackTrace();

        }

    }

PARA COPIAR UN ARCHIVO

try (

FileInputStream in = new FileInputStream(source);

FileOutputStream out = new FileOutputStream(target)

) {

     byte[] buf = new byte[1024];

     int length;

     while ((length = in.read(buf)) > 0) {

     out.write(buf, 0, length);

     }

  }

WILDCARD

public static void imprimir(List<? extends Persona> lista) {

for(Persona p:lista) {

System.out.println(p.getNombre());

}

}