Resumen Programación 2ºEVA

Tema : ArrayList

La clase ArrayList es un objeto que actúa como lista que implementa la interfaz Collection de java. Ejemplo:

ArrayList <String> nom = new ArrayList <> (); //jkd8

ArrayList <String> nom = new ArrayList <String> (); //jdk6

Debemos importar el paquete **java.util.ArrayList;**

Funciones:

- Agregar elementos => método **add()**, ejemplo: **num.add("Sylvanus"); //o bien podemos añadirlo en una determinada posición:**
  - **num.add(pos, "Janus");**
- Eliminar elementos => método **remove()**, ejemplo: **nom.remove(pos);**
- Tamaño del Array => método **size()**, ejemplo: **nom.size();**
- Vaciar el Array => método **clear()**, ejemplo: **nom.clear();**
- Comprobar la existencia de un objeto => método **contains()**, ejemplo: **nom.contains(numBusq); //devuelve un valor boolean**
- Obtener la posición de un elemento => método **IndexOf()**, ejemplo:
  - **nom.IndexOf(objetoBusq);**
- Obtener un elemento del Array => método **get()**, ejemplo: **nom.get(pos); //devuelve la posición de memoria del Array**
- Substituir un elemento del Array => método **set()**, ejemplo: **nom.(pos, objeto);**

Iterator:

Utiliza el método **hasNext()** para comprobar si hay elementos en el Array, y **next()** para obtener el siguiente objeto del Array. Ejemplo:

```
2  public class Iterador {
3
4      public static void iterator () {
5
6          Iterator it = nom.iterator();
7
8          while (it.hasNext()) {
9
10             System.out.println(it.nex());
11         }
12     }
13 }
```

Búsqueda dicotómica y ArrayList:

```java
public class Buscar {
    public static int busqueda (ArrayList <Jugador> jugador) {
        int i = 0;
        int f = jugador.size() - 1;
        int m;
        int b2;
        String nombreBusq;

        Scanner lee = new Scanner (System.in);

        System.out.println("\n Nombre del jugador a buscar: ");
        nombreBusq = lee.next();

        while (i < f) {
            m = (i + f) / 2;
            if (nombreBusq.compareToIgnoreCase(jugador.get(m).getNombre()) == 0) {
                i = m;
                f = m;
            }
            else {
                if (nombreBusq.compareToIgnoreCase(jugador.get(m).getNombre()) > 0) {
                    f = m - 1;
                }
                else {
                    i = m + 1;
                }
            }
        }
        if (nombreBusq.compareToIgnoreCase(jugador.get(i).getNombre()) == 0) {
            b2 = i;
        }
        else {
            b2 = - 1;
        }
        return b2;
    }
}
```

Ordenación de un ArrayList:

```java
public class Ordenar {

    public static void ordenaNombre (ArrayList <Jugador> jugador) {

        int i;
        int j;
        Jugador aux;

        for (i = 0; i < jugador.size() - 1; i++) {

            for (j = (i + 1); j < jugador.size(); j++) {

                if (jugador.get(i).getNombre().compareToIgnoreCase(jugador.get(j).getNombre()) < 0) {

                    aux = jugador.get(i);
                    jugador.set(i, jugador.get(j));
                    jugador.set(j, aux);
                }
            }
        }
    }
}
```

Utilización avanzada de Clases: Polimorfismo, Herencia, Ligadura Dinámica e Interfaces

Herencia:

Permite crear un objeto de la subclase a través de una superclase, gracias a este proceso un objeto de la subclase puede comportarse como uno de la superclase.

Referencia a la superclase con super (super()-constructor, o super. - un elemento)

Se indica en la clase "hija", en su definición, ejemplo:

public class Socio extends Persona {}

La subclase puede implementar los métodos de la superclase y además definir sus propias variables además de usar las de la clase padre

```
2  public class C_Socio extends C_Empleado implements I_Dividendos {
3
4      private int participaciones;
5      private final int valor = 100;
6      private double dividendo;
7
8      public C_Socio () {}
9
10     public C_Socio (String nombre, String dni, int retencion, int sueldo, int participaciones) {
11
12         super(nombre, dni, retencion, sueldo);
13         this.participaciones = participaciones;
14     }
15
16     // Como se ve a continuacion solo se ponen los getters y setters propios de la clase "hija"
17
18     public void setParticipaciones(int participaciones) {
19         this.participaciones = participaciones;
20     }
21
22     public int getParticipaciones() {
23         return participaciones;
24     }
25 }
```

Polimorfismo:

Permite redefinir el comportamiento de un método (cambiar su cuerpo), esto **solo se puede hacer cuando el método original es abstracto**, ejemplo:

```java
public abstract class Persona {

    //Atributos, constructores, getters y setters
    ...

    public abstract float calcularNomina ();
}

public class Medico extends Persona {

    //Atributos, constructores, getters y setters
    ...

    @Override //redefinicion del metodo de la clase padre (Polimorfismo)
    public float calcularNomina() {

        float nomina;
        float complementos = 100;

        nomina = super.getSueldoBase() + complementos + (50 * trienios);
        nomina = nomina - (nomina * 0.30f);

        return nomina;
    }
}
```

Ligadura dinámica:

Consiste en guardar un objeto, en tiempo de ejecución, de la clase hija en la variable de la clase padre, para poderse dar debe existir herencia, ejemplo:

```java
//en main definimos el ArrayList o Array de la clase padre
ArrayList <Persona> pers = new ArrayList <Persona> ();

 //A la hora de dar altas y crear el objeto lo creamos usando el constructor de la clase hija
 pers.add(new Medico(dni, nombre, direccion, telefono, sueldoBase, universidad, clinicaHospital, trienios));
```

Interfaces:

Sistema utilizado para enlazar clases inconexas, se utilizan cuando no es posible enlazar una clase con otra segunda, pues en Java no se permite la herencia múltiple, pero a una clase si se le permite implementar varias interfaces. También es posible usar una interfaz como tipo de dato

```
2  public interface I_Dividendos {
3
4      public double dividendo ();
5  }
6
7  public class C_Socio extends C_Empleado implements I_Dividendos {
8
9      //Atributos, constructores, getters y setters
10     ...
11
12     @Override //redefinicion del metodo de la interfaz (Polimorfismo)
13     public double dividendo() {
14
15         dividendo = participaciones * valor;
16         dividendo = dividendo - (dividendo * 0.15);
17
18         return dividendo;
19     }
20 }
```

```
22 //podriamos implentar mas interfaces
23 public class C_Socio extends C_Empleado implements I_Dividendos, I_ejemplo2 {
24
25     //Atributos, constructores, getters y setters
26     ...
27
28     @Override //redefinicion del metodo de la primera interfaz
29     public double dividendo() {
30
31         dividendo = participaciones * valor;
32         dividendo = dividendo - (dividendo * 0.15);
33
34         return dividendo;
35     }
36
37     @Override //redefinicion del metodo de la segunda interfaz
38     public double ejemplo2() {
39
40         //cuerpo del metodo de la interfaz2
41         ...
42     }
43 }
```

Método instanceof: Obtención de métodos de la clase hija

```
2  public class Visualizar {
3
4      public static void visualizaJefes (ArrayList <C_Empleado> emp) {
5
6          System.out.println("\n      ----- Listado de Jefes: ----- ");
7
8          for (C_Empleado jefe: emp) {
9
10             if (jefe instanceof C_Jefe) {
11
12                 System.out.println("\n - DNI: " + jefe.getDni() +
13                                    "\n - Nombre: " + jefe.getNombre() +
14                                    "\n - Sueldo: " + jefe.getSueldo() +
15                                    "\n - Retencion: " + jefe.getRetencion() +
16                                    "\n - Plus: " + ((C_Jefe) jefe).getPlus() +
17                                    "\n - Sueldo neto: " + ((C_Jefe) jefe).sueldoNeto() +
18                                    "\n---------------------------------------- ");
19             }
20         }
21     }
22 }
```

Asociación: Agregación y Composición

Agregación:

```
2   //AGREGACION
3   public class Departamento {
4
5       private String nombre;
6       private String especialidad;
7       private ArrayList <Empleado> empleados;
8
9       public Departamento () {}
10
11      public Departamento (String nombre, String especialidad) {
12
13          this.nombre = nombre;
14          this.especialidad = especialidad;
15          //jdk8
16          this.empleados = new ArrayList <> ();
17          //jdk6
18          //this.empleados = new ArrayList <Empleados> ();
19      }
20
21      //getters y setters
22  }
24  public class Empleado {
25
26      private String dni;
27      private String nombre;
28      private String categoria;
29
30      public Empleado () {}
31
32      public Empleado (String dni, String nombre, String categoria) {
33
34          this.dni = dni;
35          this.nombre = nombre;
36          this.categoria = categoria;
37      }
38
39      //getters y setters
40  }
```

Composición: (El hijo no puede existir sin el padre)

```java
//COMPOSICION
public class C_Autor {

    private String dni;
    private String nombre;
    private String ciudad;
    private ArrayList <C_Libros> libros;

    public C_Autor () {}

    public C_Autor (String dni, String nombre, String ciudad) {

        this.dni = dni;
        this.nombre = nombre;
        this.ciudad = ciudad;
        this.libros = new ArrayList <> ();
        //this.libros = new ArrayList <C_Libros> ();
    }

    //getters y setters
}
public class C_Libros {

    private String isbn;
    private String titulo;
    private float precio;
    ArrayList <C_Autor> autores;

    public C_Libros () {}

    public C_Libros (String isbn, String titulo, float precio) {

        this.isbn = isbn;
        this.titulo = titulo;
        this.precio = precio;
        this.autores = new ArrayList <> ();
        //this.autores = new ArrayList <C_Autor> ();
    }

    //getters y setters
}
```

Creación y asociación:

```java
//en ambos casos crearemos igual
departamentos.add(new Departamento(nombre, especialidad));
        //no añadimos el ArrayList de la agregacion/composicion

    //realizamos la union de las clases
    departamentos.get(0).getEmpleados().add(empleado.get(0)); //agregacion
        //solo lo tenemos que hacer para la clase que contiene el ArrayList de la asociacion

    autores.get(0).getLibros().add(libros.get(0)); //composicion
    libros.get(0).getAutores().add(autores.get(0)); //composicion
        //lo tenemos que hacer para ambas clases de la asociacion
```

Borrado:

```
2   //AGREGACION
3
4   //buscamos el empleado a borrar y luego procedemos a borrar
5
6   for (i = 0; i < departamentos.size(); i++) {
7       //recorremos los departamentos
8
9       for (j = 0; j < departamentos.get(i).getEmpleados().size(); j++) {
10          //recorremos de cada departamento el ArrayList de empleados
11
12          if (departamentos.get(i).getEmpleados().indexOf(empleados.get(bEmpleado)) == 0) {
13              //Si el nombre del empleado coincide con el empleado i lo borramos del departamento
14
15              departamentos.get(i).getEmpleados().remove(empleados.get(bEmpleado));
16
17              System.out.println("\n ----- Empleado eliminado con exito del departamento: " +
18                                 departamentos.get(i).getNombre() + " ----- \n");
19          }
20      }
21  }
27  //COMPOSICION
28
29  //buscamos el libro a borrar y luego procedemos a borrar
30
31  for (i = 0; i < libros.get(bLibro).getAutores().size(); i++) {
32      //recorremos el ArrayList de autores dentro del ArrayList de libros
33
34      //borramos del ArrayList de libros dentro de autores el libro seleccionado
35      bAutor = autores.indexOf(libros.get(bLibro).getAutores().get(i));
36      autores.get(bAutor).getLibros().remove(autores.get(bAutor).getLibros().indexOf(libros.get(bLibro)));
37
38      System.out.println("\n ----- Libro eliminado con exito del autor: " + autores.get(bAutor).getNombre() + " ----- ");
39  }
40
41  libros.remove(bLibro);
42  //borramos de forma definitiva el libro
```

Anidamiento de clases:

Una clase se debe definir dentro de otra solo cuando tenga sentido en el contexto de la clase que la incluye o cuando depende de la función que desempeña la clase que la incluye.

Ejemplo: una ventana puede definir su propio cursor, en este caso la clase anidada es cursor.

```
28  public class Ventana {
29
30      //Atributos de Ventana
31
32      private class Cursor {
33
34          //Atributos y metodos de Cursor
35      }
36
37      //Metodos de Ventana
38  }
```

La clase anidada recibe el nombre de **clase interna**, por ser miembro (**se denomina miembro a los atributos y métodos de la clase**) del objeto

Listas Lineales, Pilas y Colas:

Listas Lineales:

"Clase Objeto":

```java
public class Alumno {

    private String nombre;
    private String modulo;
    private int nota;
    private Alumno siguiente;

    public Alumno () {}

    public Alumno (String nombre, String modulo, int nota, Alumno siguiente) {

        this.nombre = nombre;
        this.modulo = modulo;
        this.nota = nota;
        this.siguiente = siguiente;
    }

    //getters y setters de todos los atributos, incluyendo los de siguiente

    public void setSiguiente(Alumno siguiente) {
        this.siguiente = siguiente;
    }

    public Alumno getSiguiente() {
        return siguiente;
    }
}
```

Inserción:

```java
//en main iniciamos p
Alumno p = null;

public class Altas {

    public static Alumno altaManual (Alumno p) {

        Alumno q; //p es el puntero (primero) y q es el que estamos creando
        Alumno actual;
        Alumno anterior;
        int nota;
        int i = 0;
        char decision;
        int b;

        Scanner lee = new Scanner (System.in);

        do {

            q = null;
            actual = p;
            anterior = p;

            //peticion de datos

            if (p == null) { //Creacion del primer objeto para ello p = null

                p = new Alumno(nombre, modulo, nota, q);
            }
            else { //Creacion de forma ordenada cuando ya tenemos mas objetos

                while (actual != null & nombre.compareToIgnoreCase(actual.getNombre()) > 0) {
```

```
                anterior = actual;
                actual = actual.getSiguiente();
            }

            if (anterior == actual) { //Para insertarlo en el principio de la lista

                q = new Alumno(nombre, modulo, nota, p);
                p = q;
            }
            else { //Para insertarlo en la mitad de la lista o al final

                q = new Alumno(nombre, modulo, nota, actual);
                anterior.setSiguiente(q);
            }
        }

        System.out.println("\n Alumno " + (i + 1) + " insertado con exito \n");
        i++;

        do {
            //comprobacion para otra insercion
        }
        while (b != 1);
    }
    while (decision != 'N');

    return p;
}
}
```

Pilas o Listas LIFO (Last In First Out):

"Clase Objeto":

```
public class Articulo {

    private int codigo;
    private int unidades;
    private float precio;
    private Articulo siguiente;

    public Articulo () {}

    public Articulo (int codigo, int unidades, float precio, Articulo siguiente) {

        this.codigo = codigo;
        this.unidades = unidades;
        this.precio = precio;
        this.siguiente = siguiente;
    }

    //getters y setters de todos los atributos, incluyendo los de siguiente

    public void setSiguiente(Articulo siguiente) {
        this.siguiente = siguiente;
    }

    public Articulo getSiguiente() {
        return siguiente;
    }
}
```

Inserción:

```
1  //em main iniciamos p
2  Articulo p = null;
3
4  public class Altas {
5
6      public static Articulo altaManual (Articulo p) {
7
8          Articulo q;
9          int unidades;
10         float precio;
11         int i = 0;
12         int b;
13         char decision;
14
15         Scanner lee = new Scanner (System.in);
16
17         do {
18
19             System.out.println("\n Introduce el codigo del articulo " + (i + 1) + ": ");
20             int codigo = lee.nextInt();
21
22             System.out.println("\n Introduce el numero de unidades del articulo " + (i + 1) + ": ");
23             unidades = lee.nextInt();
24
25             System.out.println("\n Introduce el precio del articulo " + (i + 1) + ": ");
26             precio = lee.nextFloat();
27
28             q = new Articulo(codigo, unidades, precio, p);
29             p = q;
30
31             i++;
33              do {
34                  //comprobacion para otra insercion
35              }
36              while (b != 1);
37         }
38         while (decision != 'N');
39
40         return p;
41     }
42 }
```

Colas o Listas FIFO (First In First Out):

"Clase Objeto":

```java
public class Espectador {

    private int numeroAsiento;
    private String nombreEspectador;
    private String tituloPelicula;
    private Espectador siguiente;

    public Espectador () {}

    public Espectador (int numeroAsiento, String nombreEspectador, String tituloPelicula, Espectador siguiente) {

        this.numeroAsiento = numeroAsiento;
        this.nombreEspectador = nombreEspectador;
        this.tituloPelicula = tituloPelicula;
        this.siguiente = siguiente;
    }

    //getters y setters de todos los atributos, incluyendo los de siguiente

    public void setSiguiente(Espectador siguiente) {
        this.siguiente = siguiente;
    }

    public Espectador getSiguiente() {
        return siguiente;
    }
}
```

Inserción:

```
//en main iniciamos p
Espectador p = null;

public class Altas {

    public static Espectador altaManual (Espectador p) {

        char decision;
        int b;
        int i = 0;
        int numeroAsiento;
        Espectador actual = p;
        Espectador q = null;

        Scanner lee = new Scanner (System.in);

        do {

            System.out.println("\n Introduce el nombre del espectador " + (i + 1) + ": ");
            if (i != 0) {

                lee.nextLine();
            }
            String nombre = lee.nextLine();

            System.out.println("\n Introduce el titulo de la pelicula del espectador " + (i + 1) + ": ");
            String tituloPelicula = lee.nextLine();

            System.out.println("\n Introduce el numero del asiento del espectador " + (i + 1) + ": ");
            numeroAsiento = lee.nextInt();
            if (p == null) { //primera alta

                q = new Espectador(numeroAsiento, nombre, tituloPelicula, p);
                p = q;
                actual = p; /* para que al añadir el siguiente sea posible añadirlo, si no como p viene en
                               null y actual vale p nunca podriamos añadir mas en esta ejecucion */
            }
            else { //ya tenemos espectadores dados de alta

                while (actual.getSiguiente() != null) {

                    actual = actual.getSiguiente();
                }

                q = new Espectador(numeroAsiento, nombre, tituloPelicula, null);
                actual.setSiguiente(q);
            }

            i++;

            do {
                //comprobacion para otra insercion
            }
            while (b != 1);
        }
        while (decision != 'N');

        return p;
    }
}
```

Borrado (Genérico para Listas Lineales, Pilas y Colas->si no usamos raíz):

```java
public class Borrar { //Borrado correcto Jenny

    public static C_Espectador borrar (BufferedReader lee, C_Espectador p) throws IOException {

        int c = 0;
        String nombre;
        //C_Articulo q = null;
        C_Espectador anterior = p;
        C_Espectador actual = p;

        if (p == null)
            System.out.println("No hay ningun espectador registrado en la base de datos");

        else {
            Visualizar.visualizar(p); //Visualiza la lista de espectadores

            System.out.println("-----------------------------------------------------");
            System.out.println("Introduzca el nombre del espectador que desea borrar: ");
            nombre = lee.readLine();

            while (actual != null && nombre.compareToIgnoreCase(actual.getNombre()) != 0) {
                anterior = actual;
                actual = actual.getSiguiente();
            }
            if (actual == null)
                System.out.println("Espectador no encontrado");
            else {
                do {
                    Visualizar.visualizarEspectador(actual);
                    c = Menu.menuConfinrmacion(lee, c);  //Muestra un menu 1-borrar 2-salir
                   }
                } while (c < 1 || c > 2);
                if (c == 1) {
                    if (anterior == actual)
                        p = p.getSiguiente();
                    else
                        anterior.setSiguiente(actual.getSiguiente());

                    System.out.println("Espectador borrado");
                }
            }
        }
        return p;
    }
}
```

Excepciones:

Una **excepción** es un tipo de error o una condición anormal que se ha producido durante la ejecución del programa, esta puede provocar la finalización del programa

Las excepciones se clasifican siguiendo el siguiente esquema:

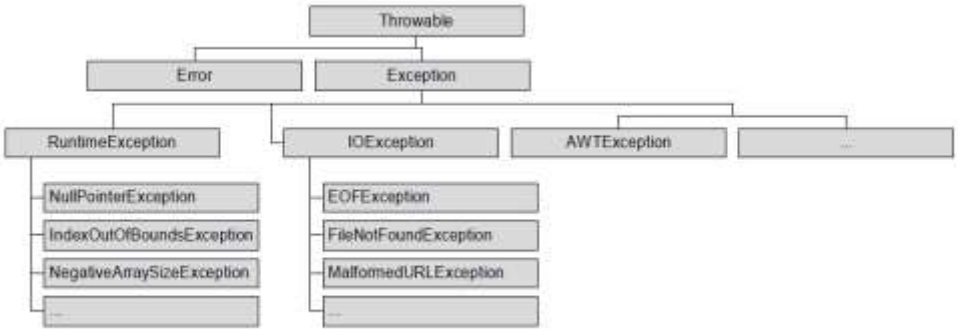

Figura 8.1: Jerarquía de clases derivadas de Throwable.

- La **clase Error** está relacionada con errores de compilación, del sistema o de la JVM, son errores irrecuperables y no dependen del programador, no se deben capturar no tratar
- La **clase Exception** se divide en varias subclases, trataremos las más destacadas:
  - **Excepciones implícitas (solo existe la siguiente y sus derivadas):**
    - **RuntimeException:** Excepciones frecuentes relacionadas con errores de programación, no necesita bloques try/catch. Algunos ejemplos:

Clase	Situación de excepción
ArithmeticException	Cuando ocurre una operación aritmética errónea, por ejemplo división entre cero; con los valores reales no se produce esta excepción.
ArrayStoreException	Intento de almacenar un valor de tipo erróneo en una matriz de objetos
IllegalArgumentException	Se le ha pasado un argumento ilegal o inapropiado a un método
IndexOutOfBoundsException	Cuando algún índice (por ejemplo de array o String) está fuera de rango
NegativeArraySizeException	Cuando se intenta crear un array con un índice negativo
NullPointerException	Cuando se utiliza como apuntador una variable con valor null

  - **Excepciones explícitas (Es necesario tenerlas en cuenta y chequear cuando se producen):**
    - **IOException**
    - **AWTException**

Las clases derivadas de Exception pueden pertenecer a distintos packages, por heredar de Throwable todos los tipos de excepciones pueden usar los siguientes métodos:

- **String getMessage()** Extrae el mensaje asociado a la excepción
- **String toString()** Devuelve un String que describe la excepción
- **void printStackTrace()** Indica el método donde se lanzó la excepción

Lanzar una excepción:

Para lanzar una excepción debemos crear un objeto Exception de la clase adecuada y lanzar la excepción con la sentencia **throw** seguida del objeto Exception creado. Cuando esto sucede el método termina de inmediato (sin devolver ningún valor), a no ser que este incluido en un bloque try/catch. Ejemplo sencillo:

```
15  //Código que lanza la excepción MyException una vez detectado el error
16  MyException me = new MyException("MyException message");
17  throw me;
```

Más completo en el apartado **Excepciones del programador**

Capturar una excepción:

Para ello usamos los bloques try/catch, en el bloque try se introduce el código susceptible de causar excepciones y en el bloque catch controlamos las situaciones de excepción, entre ambos bloques no puede haber ninguna instrucción, ejemplo:

```
1  do {
2      try {
3          Visualizar.visualizarEspectador(actual);
4          c = Menu.menuConfinrmacion(lee, c);  //Menu que muestra 1-Borrar 2-Salir
5      }
6      catch (NumberFormatException e1){
7          System.out.println("'Error', no introduzca letras solo numeros porfavor");
8      }
9  } while (c < 1 || c > 2);
```

Podemos poner varios bloques catch que comprobaran las excepciones de la más concreta a la más genérica, por ejemplo:

```
1  try {
2      //instrucciones susceptibles de causar cierto tipo de excepciones
3  }
4  catch (NullPointerException e) { //MAS CONCRETO
5      /* instrucciones que se deben ejecutar si ocurre la excepcion
6         de tipo NullPointerException */
7  }
8  catch (RuntimeException e) {
9      /* instrucciones que se deben ejecutar si ocurre la excepcion
10        de tipo RunTimeException */
11 }
12 catch (Exception e) { //MAS GENERICO
13     /* instrucciones que se deben ejecutar si ocurre la excepcion
14        de tipo Exception */
15 }
```

Los bloques catch se ejecutarán si...:

1. Si la excepción es tipo NullPointerException se ejecutarán las instrucciones del 1ºbloque catch
2. Si la excepción es de tipo RunTimeException (pero no NullPointerException) se ejecutarán las instrucciones del 2ºbloque catch
3. Si la excepción es genérica (Exception, pero distinta de RunTimeException y sus derivadas) se ejecutarán las instrucciones del 3ºbloque catch

Relanzar una excepción o Propagación de excepciones:

Se produce cuando dentro de un bloque try llamamos a otro método que llama (o no) a otros métodos, en los que se puede dar la excepción, esta puede ser (o no) tratada en ese propio método, en caso de no poderla tratar se tratara en el primer método llamante que pueda tratarla

Para que los métodos llamados que no traten la excepción puedan devolvérsela al método que los llamó, deben tener en su definición **throws tipoExcepción1, tipoExcepción2**, por ejemplo:

```
19  public static void metodoLlamado() throws IOException, MyException {
20      // Código que puede lanzar las excepciones IOException y MyException
21      ...
22      /* En caso de producirse una exececpcion se le delvovera al metodo
23         anterior para que lo trate (propagandose hacia atras hasta ser tratada */
24  }
```

Bloque finally:

Último bloque del conjunto try/catch, es el encargado de devolver al sistema al estado anterior a la ejecución del programa, se usa para cerrar ficheros que han quedado abiertos, cerrar comunicaciones incompletas,...

Un ejemplo básico:

```
1   try {
2       //instrucciones susceptibles de causar excepciones
3   }
4   catch (TipoExcepcion1 Identificador) {
5       //instrucciones del 1ºbloque catch
6   }
7   catch (TipoExcepcionN Identificador) {
8       //instrucciones del 2ºbloque catch
9   }
10  finally {
11      /* instrucciones que siempre se ejecutan, se produzca
12         o no una excepciones */
13  }
```

Excepciones del programador:

Lo más adecuado es que las excepciones sean objetos, por esta razón tendremos una "clase objeto" como la siguiente:

```java
public class MisExcepciones extends Exception {

    private String error;

    public MisExcepciones () {}

    public MisExcepciones (String message) {

        super(message);
        this.error = message;
    }

    public String getError() {
        return error;
    }
}
```

Una vez construida la clase con la que levantaremos nuestras excepciones podemos empezar a definir las excepciones, para ello solo debemos crear una clase en la que controlemos los errores que deseemos por ejemplo:

```java
public class Validar {

    public static int validarDNI(String dni) {
        int error = 0;

        try {
            if (dni.length() != 9) {
                error = 1;
                throw new MisExcepciones(" Longitud del dni incorrecta, deben ser 9 caracteres ");
            }
            if (!dni.substring(0, 8).matches("[0-9]*")) {
                error = 1;
                throw new MisExcepciones(" Los 8 primeros caracteres deben ser numericos ");
            }
            if (!dni.substring(8).matches("[A-Za-z]")) {
                error = 1;
                throw new MisExcepciones(" El 9º caracter debe ser una letra ");
            }
            String letras = "TRWAGMYFPDXBNJZSQVHLCKE";
            String letraComp = dni.substring(8);
            int numDNI = Integer.parseInt(dni.substring(0, 8));

            int posLetra = numDNI % 23;
            if (letraComp.toUpperCase().charAt(0) != letras.charAt(posLetra)) {
                error = 1;
                throw new MisExcepciones(" La letra no es correcta, deberia ser: " + letras.charAt(posLetra));
            }
        }
        catch (MisExcepciones e) {
            System.out.println(e.getError());
        }
        return error;
    }
```

```java
    public static int validarNombre (String nombre) {
        int error = 0;

        try {
            if (nombre.length() > 20) {
                error = 1;
                throw new MisExcepciones(" El nombre no puede superar los 20 caracteres ");
            }
            if(!nombre.matches("[A-Za-z]*")) {
                error = 1;
                throw new MisExcepciones(" El nombre no puede contener numeros ");
            }
        }
        catch (MisExcepciones e) {
            System.out.println(e.getError());
        }
        return error;
    }
    public static int validarTelefono (String telefono) {
        int error = 0;

        try {
            if (telefono.length() != 9) {
                error = 1;
                throw new MisExcepciones(" El telefono debe tener 9 caracteres ");
            }
            if (!telefono.matches("[0-9]*")) {
                error = 1;
                throw new MisExcepciones(" El telefono solo puede contener numeros ");
            }
        }
        catch (MisExcepciones e) {
            System.out.println(e.getError());
        }
        return error;
    }
}
```

Para usar nuestra excepción (validación) debemos llamar al método pasándole el dato a validar, ejemplo:

```
//DURANTE LAS ALTAS COMPROBAMOS LOS DATOS: DNI, NOMBRE Y TELEFONO
do {
    System.out.println("\n DNI del medico " + (i + 1) + ": ");
    dni = lee.next();
    error = Validar.validarDNI(dni);
}
while (error == 1);

b = 0;
do {

    System.out.println("\n Nombre del medico " + (i + 1) + ": ");
    if (b == 0) {
        lee.nextLine();
    }
    nombre = lee.nextLine();
    error = Validar.validarNombre(nombre);

    if (error == 1) {
        b = 1;
    }
}
while (error == 1);

do {
    System.out.println("\n Telefono del medico " + (i + 1) + ": ");
    telefono = lee.nextLine();
    error = Validar.validarTelefono(telefono);
}
while (error == 1);
```

Métodos complementarios a los temas anteriores:

Llamar al constructor THIS:

A diferencia del resto de métodos de clase, el constructor no puede ser invocado directamente, pero lo podemos invocar de forma indirecta usando el método **this**

La llamada a un constructor solo puede realizarse desde otro constructor de su misma clase y siempre debe aparecer como primera sentencia, ejemplo :

```
1  public Fecha (int dd) {
2
3      this(); //invoca al constructor sin parametros
4
5      dia = dd;
6      if (!fechaCorrecta()) {
7          System.out.println("Fecha incorrecta. Se asigna la actual");
8          asignarFecha();
9      }
10 }
```

Métodos estáticos:

Atributo static:

Es un atributo asignado a las variables de clase, nunca puede asignarse a un atributo de objeto. Su principal función es que solo se le asigna una vez un espacio de memoria, mientras que a las variables de objeto se les asigna un espacio de memoria cada vez que se instancia.

Acceso a los atributos static:

Un método static puede acceder a todos los miembros static (atributo o método) de su clase, pero no puede acceder a los no estáticos

Iniciador estático:

Es un método anónimo que no tiene parámetros, no retorna ningún valor, y es invocado automáticamente por el sistema cuando se carga la clase, ejemplo:

```java
public class Circulo { //Ejemplo de Ceballos

    //Atributos
    private static double pi = 3.14;
    public static numCirculos;
    public static double seno[] = new double[360];
    public static double coseno[] = new double[360];

    //iniciador estatico
    static {

        //tablas del seno y coseno de grado en grado
        for (ini i = 0; i < 360; i++) {

            double s. c:
            //calcular el seno y el coseno de i
            s = Math.sin(Math.toRadians(i));
            c = Math.cos(Math.toRadians(i));

            //almacenar los valores redondeados a 6 decimales
            seno[i] = Math.rint(s*1000000)/1000000:
            coseno[i] = Math.rint(c*1000000)/1000000:
        }
    }
    //resto del codigo
}
```

Métodos static:

Un método declarado como estático carece de referencia **this** por lo que no puede ser invocado para un objeto de su clase, sino que se invoca en general allí donde se necesite.

Final:

final es una palabra clave en Java utilizada para restringir algunas funcionalidades. Podemos declarar variables, métodos y clases con la palabra clave final.

Tan poderoso y útil como la anulación de método (method overriding) y la herencia, a veces querrás evitarlos. Por ejemplo, puede tener una clase que encapsule el control de algún dispositivo de hardware. Además, esta clase podría ofrecer al usuario la capacidad de inicializar el dispositivo, haciendo uso de información privada y de propiedad. En este caso, no desea que los usuarios de su clase puedan anular el método de inicialización.

Cualquiera que sea el motivo, en Java es fácil evitar que un método sea anulado o que una clase se herede usando la palabra clave final.

Uso final para evitar anulación de métodos (Overriding)

Para evitar que un método sea anulado, especifique como un modificador al comienzo de su declaración. Los métodos declarados como **final** no pueden anularse. El siguiente fragmento ilustra el uso de *final*:

```
class A {
        final void metodo(){
        System.out.println("Este es un método final.");
        }
}
class B extends A {
       void metodo(){
              //ERROR
              // 'metodo()' cannot override 'metodo()' in 'A';
              // overridden method is final
              System.out.println("Ilegal!");
       }
}
```

Como ***metodo()*** se declara como final, **no** se puede **sobrescribir** en B. Si intentas hacerlo, se producirá un error en tiempo de compilación.

Uso de final para evitar herencia

Puede evitar que una clase se herede precediendo a su declaración con final.

Declarar una clase como final implícitamente declara también todos sus métodos como finales. Como era de esperar, **es ilegal declarar una clase como abstracta y final** ya que una clase abstracta es incompleta por sí misma y depende de sus subclases para proporcionar implementaciones completas. Aquí hay un ejemplo de una clase final:

```
final class A {
//...
}
//La siguiente clase es ilegal
class B extends A {
//ERROR
}
```

Como se lee en los comentarios, es ilegal que B herede A ya que A se declara como **final**.

Usar final con los miembros de datos

Además de los usos de que acabamos de mostrar, también se puede aplicar a las variables miembro para crear lo que equivale a constantes con nombre. Si antecede el nombre de una variable de clase con final, su valor no se puede cambiar a lo largo de la vida útil de su programa.

Por supuesto, puede darle a esa variable un valor inicial. Por ejemplo, el siguiente ejemplo muestra una clase simple de gestión de errores llamada *MsgError*. Esa clase mapea una cadena legible a un código de error. Aquí, esa clase original contiene constantes finales que representan los errores. La idea es que en vez de pasarle a *getMsgError()* un número como 2, puede pasar la constante entera nombrada *DOS*.

```
//Retornando un objeto String
class MsgError {
//Códigos de Error
final int CERO =0;
final int UNO = 1;
final int DOS = 2;
final int TRES = 3;
String msgs[]={"ERROR CERO","ERROR UNO","ERROR DOS","ERROR TRES"};
//Retornando un mensaje de error
        String getMsgError(int i){
                if (i>=0&i<msgs.length)
                return msgs[i];
                else
                return "CÓDIGO DE ERROR INVÁLIDO";
        }
}
class FinalD {
          public static void main(String[] args) {
          MsgError error=new MsgError();
          System.out.println(error.getMsgError(error.CERO));
          System.out.println(error.getMsgError(error.DOS));
          }
}
```

Salida:

```
ERROR CERO
ERROR DOS
```

Observe cómo las constantes finales se usan en . Como son miembros de la clase *MsgError*, se debe acceder a ellos a través de un objeto de esa clase. Por supuesto, también pueden ser heredados por subclases y acceder directamente dentro de esas subclases.

Destructor: Método finalize

Un objeto es destruido automáticamente cuando se eliminan todas las referencias al mismo (cuando el flujo de ejecución salga fuera del ámbito donde está declarada o porque se le asigna valor null)

Un **destructor** es un método especial de una clase que se ejecuta antes de que un objeto de esa clase sea eliminado físicamente de la memoria, se indica mediante: **finalize**. Cuando en una clase no es especificado el compilador le asigna uno a través de la clase **Object** con la siguiente sintaxis: **protected void finalize() throws Throwable {}**

Solo es posible definir un destructor y debe tener la sintaxis anterior, salvo que podemos definir en el cuerpo las instrucciones que deseemos, por ejemplo una visualización (**System.out.println("Objeto destruido");**)

El método destructor siempre es invocado antes de recolocar un objeto como basura por el recolector de basura de Java, cuando no queden referencias del objeto

El destructor también puede llamar explícitamente: **objeto.finalize();**

Recolector de basura:

El recolector de basura se ejecuta en un subproceso paralelamente a su aplicación limpiando objetos desreferenciados en forma silenciosa y en segundo plano y nunca se detiene

Si deseamos que se realice de forma completa la recolección de basura (pues es posible que no se marquen todos los objetos desreferenciados antes de la finalización del programa) debemos forzarlo llamando al método **gc** (**garbage collector**) de la clase **System**

Aclaraciones:

Sobrecarga vs. Polimorfismo

Método **sobrecargado**

- mismo nombre y tipo retorno,
- distinto número, tipo de parámetros u orden de los mismos

Polimorfismo

Sobreescritura de métodos.

- Que un mismo método, con el mismo nombre y argumentos,
- tengan un cuerpo diferente cuando entre la clase donde están definidos. Están relacionadas dichas clases por herencia o implementación de una interfaz

Static vs. Final

Static usado con objeto de no tener que instanciar el objeto.
Final esta palabra en la declaración, no permite modificación/herencia.

Ej. Dentro del método main, al ser este un método de clase y llevar incorporado **static** en su declaración no podemos declarar constantes de clase (variables globales).

public static void main (String[] args)

```
class Ideone
{
   public static void main (String[] args) throws java.lang.Exception
   {
      final double PI = 3.1416;//no se puede usar static
   }
}
```

```
class Ideone
{
   static final double PI = 3.1416;

   public static void main (String[] args) throws java.lang.Exception
   {
      System.out.println(Ideone.PI);
   }
}
```

Definiciones varias:

- **Herencia:** relación entre clases "padre" y clases "hijas" donde la clase hija hereda/tiene los atributos y métodos de clase padre. Aplicándole polimorfismo a los métodos se le puede dar otro cuerpo y/o instrucciones
- **Polimorfismo:** Sobreescritura de métodos. Que un mismo método, con el mismo nombre y argumentos, tengan un cuerpo diferente cuando entre la clase donde están definidos están relacionados por herencia o implementación de una interfaz
- **Ligadura dinámica:** Cuando en tiempo de ejecución guardamos en una variable de tipo padre un objeto de tipo hijo
- **Interfaz:** Es una estructura/protocolo donde indicamos unas constantes y declaramos unas constantes y/o métodos abstractos. Con ella podemos relacionar clases que no tienen otro tipo de relación (herencia, agregación y composición), implementándola en las dos clases
- **Clase:** Estructura/protocolo donde declaramos variables globales y/o métodos/funciones dándoles un cuerpo/instrucciones a los métodos
- **Objetos:** Instancia de una clase // Variable de referencia cuyo tipo es una clase, esta clase puede ser una clase de la API de Java o una
    - construida por el programador
- **Atributo:** Característica/variable propia de un objeto que al instanciar le asignamos un valor
- **Variable global:** Variable que se puede utilizar en todos los métodos propios de la clase, se indica con la palabra reservada **static** que es propia de la clase y no del objeto
- **Clase abstracta:** Clase en la cual existe un método abstracto, esto es, que no le damos cuerpo, y esta clase no la podemos instanciar, es decir, no podemos crear objetos de esta clase **Diferencias entre interfaz y clase abstracta:**
    - Los métodos de la interfaz son todos abstractos, mientras que en la clase abstracta solo es necesario que tenga un solo método abstracto
    - En la interfaz declaramos constantes, mientras que en la clase abstracta declaramos variables
    - La interfaz tiene de tipo de dato interfaz, mientras que la clase tiene de tipo de dato una clase
    - En la clase abstracta puede haber una implementación de herencia de otra clase e implementar interfaces, mientras que las interfaces solo pueden heredar la implementación de otra interfaz
    - Los métodos y constantes de la interfaz no pueden ser privados, mientras que los de las clases abstractas pueden tener métodos y variables privadas