

Java Inicial

(20 horas)





Temario

1. Programación Orientada a Objetos
2. Introducción y Sintaxis Java
3. Sentencias Control Flujo
4. POO en Java
5. Relaciones entre Objetos
6. Polimorfismo, abstracción e interfaces
7. **Excepciones**
8. Conceptos avanzados



Tema 7

Excepciones y Organización de Paquetes



Objetivos

1. Programación Orientada a Objetos
2. Introducción y Sintaxis Java
3. Sentencias Control Flujo
4. POO en Java
5. Relaciones entre Objetos
6. Polimorfismo, abstracción e interfaces
7. **Excepciones**
8. Conceptos avanzados

■ Excepciones

- Gestión excepciones
- Jerarquía
- Captura
- Creación
- Lanzar excepciones

■ Paquetes

- Organización
- Creación e inclusión
- JAVA API
- Consideraciones

■ Excepciones

□ Debemos controlar los errores

```
ABRIR_FICHERO_LECTURA( unFichero )
MIENTRAS NO FINAL_FICHERO
    LEER_DATO( unFichero )
    PROCESAR_DATO
    ESCRIBIR_RESULTADO
FIN_MIENTRAS
CERRAR_FICHERO( unFichero )
```

```
SI_ERROR_APERTURA
    TRATAR_ERROR_APERTURA
SI_ERROR_LECTURA
    TRATAR_ERROR_LECTURA
SI_ERROR_PROCESO
    TRATAR_ERROR_PROCESO
SI_ERROR_ESCRITURA
    TRATAR_ERROR_ESCRITURA
SI_ERROR_CIERRE
    TRATAR_ERROR_CIERRE
```

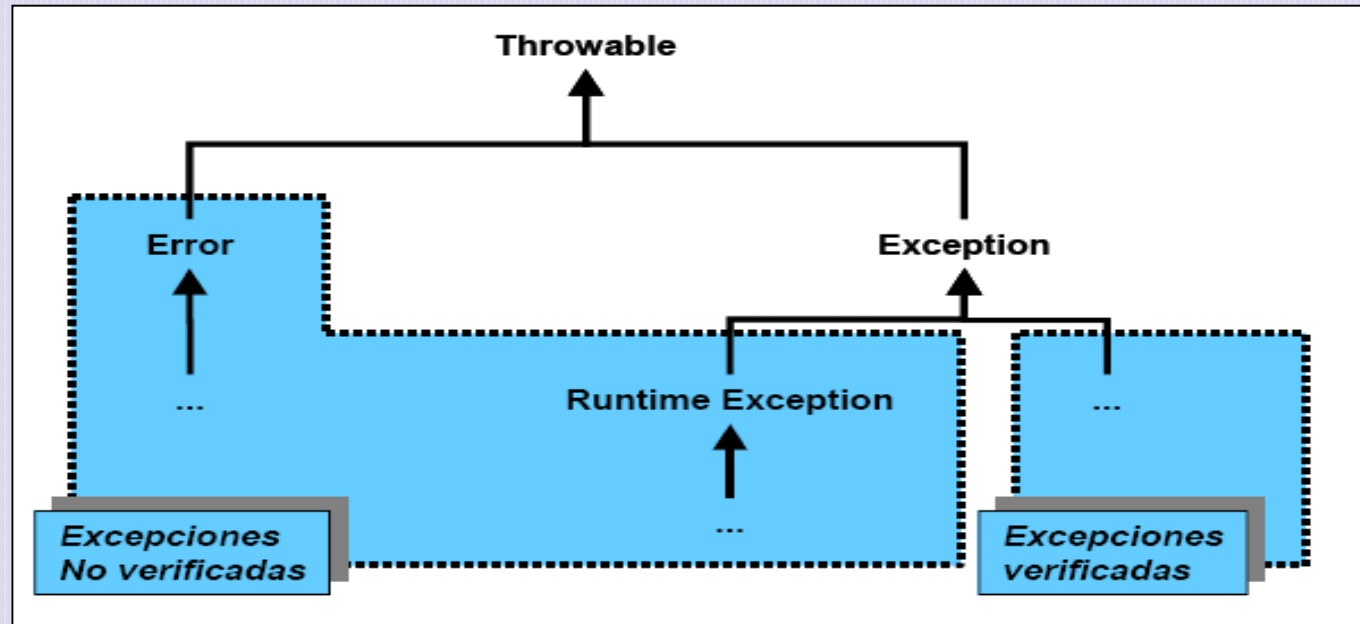
Código

Gestión de errores

■ Gestión de Excepciones

- Nos permiten realizar nuestras sentencias independientemente de si producen errores
- Podemos tratar los errores originados
- Las excepciones son objetos Java
 - Se pueden definir
 - Están jerarquizadas
 - Son heredables

■ Jerarquía de excepciones



□ No verificadas:

- Se pueden producir en cualquier punto de nuestra
- No se obliga a que sean controladas

□ Verificadas:

- Se obliga a realizar un tratamiento de la excepción

■ Captura de Excepciones

- **Try:** Indica que las sentencias de su bloque pueden producir algún tipo de excepción.
- **Catch:** Este bloque captura las excepciones producidas en el *try*.

```
try{
    //sentencias que pueden provocar excepciones
}
catch(TipoExcepcion1 id){
    // Sentencias a realizar cuando se produzca esa excepción
}
catch(TipoExcepcion2 id){
    // Sentencias a realizar cuando se produzca esa excepción
}
...
[finally{
    //Sentencias a realizar se produzca o no la excepción
}]
```

■ Excepciones: Ejemplos

□ División por cero

```
class TestExcepciones
{
    public static void main(String args[])    {
        System.out.println ("La division de " + args[0] + " y " +
            args[1] + " es ");
        System.out.println (Integer.parseInt(args[0]) /
            Integer.parseInt(args[1]));
    }
}
```

La salida generada al ejecutar será:

```
java TestExcepciones 6 2
La division de 6 y 2 es 3
```

Pero que ocurre si lo ejecutamos introduciendo como divisor el 0:

```
java TestExcepciones 6 0
La division de 6 y 0 es
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at TestExcepciones.main(TestExcepciones.java:7)
```

Y si nos olvidamos de introducir un argumento:

```
java TestExcepciones 6
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
    at TestExcepciones.main(TestExcepciones.java:5)
```

■ Excepciones: Ejemplos (II)

- ArithmeticException : al dividir por 0
- NumberFormatException: operandos no es numérico
- ArrayIndexOutOfBoundsException: faltan operandos

```
class TestExcepciones
{
    public static void main(String args[])
    {
        try
        {
            System.out.println ("La division de " + args[0] + " y " +
                args[1] + " es ");
            System.out.println (Integer.parseInt(args[0]) /
                Integer.parseInt(args[1]));
        }
    }
}
```

La salida en caso de que el segundo operando sea un 0

```
La division de 3 y 0 es
No se puede dividir por 0
```

La salida en caso de que algún operando no sea numérico

```
La division de 3 y uno es
Algun operando no es un numero
```

La salida en caso de no introducir los dos operandos necesarios

```
Falta algun operando
```

```
}
}
```

■ Excepciones: Ejemplos (III)

- En el caso de querer controlar cualquier excepción sin importar el tipo

```
class TestExcepciones
{
    public static void main(String args[])
    {
        try
        {
            System.out.println ("La division de " + args[0] + " y " +
                                args[1] + " es ");
            System.out.println (Integer.parseInt(args[0]) /
                                Integer.parseInt(args[1]));
        }
        catch (Exception e)
        {
        }
    }
}
```

La salida en cualquiera de los casos que pueden provocar error será la siguiente:

Error de ejecución

```
}
}
}
```

■ Excepciones: Ejemplos (III)

□ Podemos obtener el mensaje *getMessage()*

```
class TestExcepciones
{
    public static void main(String args[])
    {
        try
        {
            System.out.println ("La division de " + args[0] + " y " +
                                args[1] + " es ");
            System.out.println (Integer.parseInt(args[0]) /
                                Integer.parseInt(args[1]));
            ...
        }
        catch (Exception e)
        {
            System.out.println ("Error de ejecucion: " + e.getMessage());
        }
        ...
    }
}
```

■ Excepciones: Ejemplos (III)

- Podemos obtener el mensaje *getMessage()*

```
class TestExcepciones
```

Las salidas serían las siguientes:

```
La division de 3 y 0 es
```

```
Error de ejecucion: / by zero
```

cuando intentamos dividir por 0

```
La division de 3 y uno es
```

```
Error de ejecucion: uno
```

cuando introducimos un operando que no es numérico

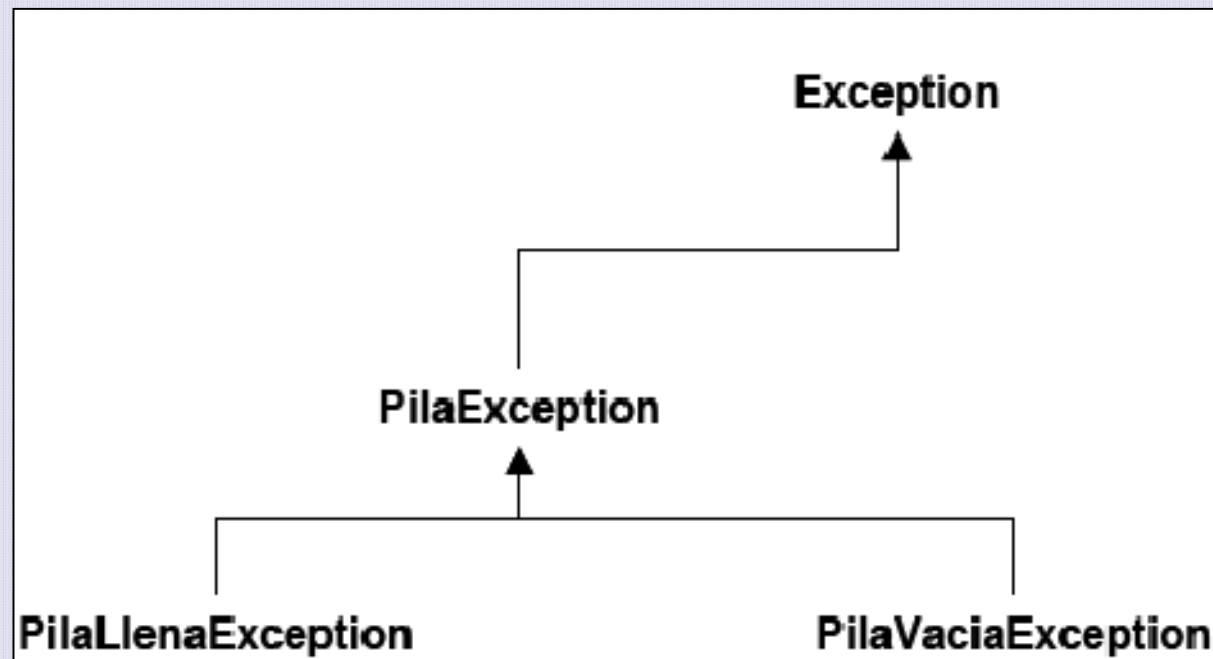
```
Error de ejecucion: null
```

y cuando nos olvidamos de algún parámetro

```
}  
}
```

■ Crear Excepciones

- Como programadores podemos crear nuestras propias excepciones
- Deben heredar de la clase ***Exception***
- Podemos realizar jerarquías de Excepciones



■ Ejemplo creación jerarquía

```
public class PilaException extends Exception
{
    public PilaException(){
        super("ERROR: fallo en la pila");
    }
    public PilaException(String msj) {
        super(msj);
    }
}
public class PilaLlenaException extends PilaException
{
    public PilaLlenaException(){
        super("ERROR: La pila esta LLENA");
    }
    public PilaLlenaException(String msj) {
        super(msj);
    }
}
public class PilaVacíaException extends PilaException
{
    public PilaVacíaException(){
        super("ERROR: La pila esta VACIA");
    }
    public PilaVacíaException(String msj) {
        super(msj);
    }
}
```

```
graph TD
    PilaException["PilaException extends Exception"]
    PilaLlenaException["PilaLlenaException extends PilaException"]
    PilaVacíaException["PilaVacíaException extends PilaException"]
    PilaLlenaException --> PilaException
    PilaVacíaException --> PilaException
```

■ Lanzar excepciones

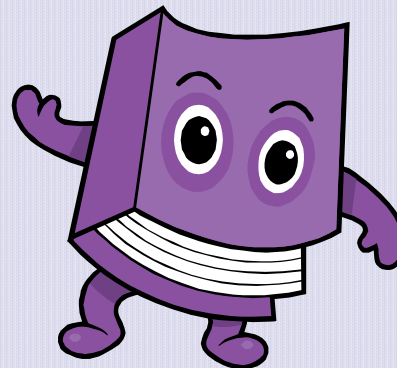
- El propio sistema puede lanzarlas por si mismo (XJ: División por cero)
- Podemos lanzarlas programáticamente
 - ***throw new TipoExcepcion()***
 - *Creamos un objeto de tipo Excepción a través de su constructor*
- Debemos indicar en el método que puede provocar la excepción

```
public void metodo() throws TipoExcepcion
{
    ...
    throw new TipoExcepcion();
    ...
}
```

■ Ejercicio práctico: Una Pila

- Implementar una clase que simule el comportamiento de un pila
 - Si la pila está llena y queremos apilar un nuevo elemento deberá provocar una excepción
 - Si la pila está vacía y se desea desapilar un elemento, deberemos generar una excepción

■ Suerte!!!



■ Solución: Una Pila



```
public class Pila
{
    private int elementos[];
    private int numElementos;
    public Pila()    {
        this(5);
    }
    public Pila(int n) {
        elementos = new int[n];
    }
    public boolean estaLlena(){
        return numElementos == elementos.length;
    }
    public boolean estaVacía(){
        return numElementos==0;
    }
    public int desapilar() throws PilaVacíaException{
        if(this.estaVacía()) throw new PilaVacíaException();
        numElementos--;
        return elementos[numElementos];
    }
    public void apilar(int e) throws PilaLlenaException{
        if(this.estaLlena())    throw new PilaLlenaException();
        elementos[numElementos] = e;
        numElementos++;
    }
    public void vaciar() throws PilaVacíaException{
        int aux;
        while(!this.estaVacía())
            aux = this.desapilar();
    }
    public int getNumElementos(){
        return numElementos;
    }
}
```

■ Solución: Una Pila (II)



```
public class TestPila
{
    public static void main(String[] args)
    {
        Pila pila1 = new Pila(3);
        for (int i = 0; i<4; i++)
        {
```

La salida obtenida sería la siguiente:

```
Se va a apilar el 0
Se va a apilar el 2
Se va a apilar el 4
Se va a apilar el 6
ERROR: La pila esta LLENA
Elemento desapilado: 4
Elemento desapilado: 2
Elemento desapilado: 0
ERROR: La pila esta VACIA
```

```
try
{
    System.out.println ("Elemento desapilado: " +
pila1.desapilar());
}
catch(PilaVacíaException e){
    System.out.println (e.getMessage());
}
}
}
```

■ Ejercicio práctico: Atracción de Feria

Diseñar una aplicación que controle el siguiente funcionamiento:

Se tiene una atracción de feria en la cual se paga un seguro de accidentes. La aseguradora se hará cargo en caso de accidente siempre que la persona que haya querido entrar en la atracción tenga una altura comprendida entre los 150 cms y los 190 cms.

Para ello se dispone de un sensor en la entrada que detecta la altura de la persona que desea montar en la atracción. Dicho sensor será una clase que dispondrá de un único método obtenerAltura() que devolverá un entero (los cms que mide la persona)

Esa altura se pasará a un sistema que validará la altura, abriendo la puerta para aquellas personas que tengan la altura requerida. En cualquier otro caso se generará un error que posteriormente será tratado.

Se desea que los errores puedan ser diferenciados (para dejar constancia de ellos) por las siguientes causas:

Por un error del sensor se puede obtener una altura negativa.

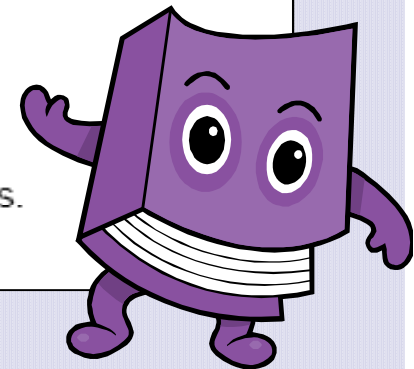
Se ha excedido la altura máxima permitida

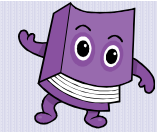
No se alcanza la altura mínima permitida

Diseñar la clase Sensor y PuertaEntrada, así como las excepciones necesarias.

Las clases de excepciones creadas son las siguientes:

■ Suerte!!!





■ Solución: Atracción de Feria

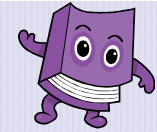
```
public class AlturaException extends Exception{
    public AlturaException(){
        super("ERROR: Altura no permitida");
    }
}

public class AlturaNegativaException extends AlturaException{
    public AlturaNegativaException(){
        super("ERROR: Altura negativa NO PERMITIDA");
    }
}

public class AlturaMinimaException extends AlturaException{
    public AlturaMinimaException(){
        super("ERROR: Altura minima no alcanzada");
    }
}

public class AlturaMaximaException extends AlturaException{
    public AlturaMaximaException(){
        super("ERROR: altura maxima sobrepasada");
    }
    public AlturaMaximaException(String msj) {
        super(msj);
    }
    public AlturaMaximaException(int maxima) {
        super("ERROR: altura maxima de " + maxima + " sobrepasada");
    }
}
```

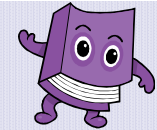
■ Solución: Atracción de Feria (II)



□ Clase Sensor

```
import java.util.Random;
public class Sensor{
    private int rangoMin, rangoMax;
    private Random lector;
    public Sensor(int min, int max) {
        rangoMin = min;
        rangoMax = max;
        lector = new Random();
    }
    public int obtenerAltura(){
        /**Si el minimo es 10 y el maximo es 20 generará un n°
        * aleatorio entre 0 y 20-10 a cuyo valor le sumaremos 10*/
        int valor = lector.nextInt(rangoMax-rangoMin) + rangoMin;
        System.out.println ("SENSOR: Leyendo altura de "+ valor);
        return valor;
    }
}
```

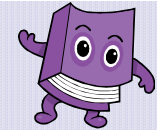
■ Solución: Atracción de Feria (III)



□ Clase Puerta de Entrada

```
public class PuertaEntrada
{
    private int alturaMinima, alturaMaxima;
    public PuertaEntrada(int hMin, int hMax)
    {
        alturaMinima = hMin;
        alturaMaxima = hMax;
    }
    public void chequear(int altura) throws AlturaNegativaException,
        AlturaMinimaException, AlturaMaximaException
    {
        if (altura < 0)
            throw new AlturaNegativaException();
        if (altura < alturaMinima)
            throw new AlturaMinimaException(alturaMinima);
        if(altura > alturaMaxima)
            throw new AlturaMaximaException(alturaMaxima);
        System.out.println ("Abriendo la puerta para "+altura+" cms");
    }
}
```

■ Solución: Atracción de Feria (III)



□ Clase de Test

```
public class TestAtraccion
{
    public static void main
    {
        Sensor s = new Senso
        PuertaEntrada puerta
        for (int i = 0; i<20
        {
            try
            {
                puerta.chequea
            }
            catch (AlturaExcep
            {
                System.out.pri
            }
            System.out.printl
        }
    }
}
```

La salida obtenida sería:

```
SENSOR: Leyendo altura de 188
Abriendo la puerta para 188 cms

SENSOR: Leyendo altura de 219
ERROR: altura maxima de 190 sobrepasada

SENSOR: Leyendo altura de 206
ERROR: altura maxima de 190 sobrepasada

SENSOR: Leyendo altura de 187
Abriendo la puerta para 187 cms

SENSOR: Leyendo altura de 170
Abriendo la puerta para 170 cms

SENSOR: Leyendo altura de 122
ERROR: Altura minima de 150 no alcanzada

SENSOR: Leyendo altura de 120
ERROR: Altura minima de 150 no alcanzada

SENSOR: Leyendo altura de 130
ERROR: Altura minima de 150 no alcanzada

SENSOR: Leyendo altura de 174
Abriendo la puerta para 174 cms

SENSOR: Leyendo altura de 209
ERROR: altura maxima de 190 sobrepasada
```

- Organización de Clases: Paquetes
 - Permite un desarrollo más eficiente de nuestras aplicaciones.
 - Las clases se almacenarán siguiendo una correcta clasificación.
 - Esta clasificación se comportará como una librería permitiendo reutilizar código de una manera más clara.

```
import java.util.StringTokenizer;
class MiClase
{
    ...
    StringTokenizer str = new StringTokenizer();
    ...
}
```

■ Creación de Paquetes

- Podemos crear nuestros paquetes ***package***

```
package paquete1;  
public class MiClase  
{  
    ...  
}
```

- Utilizar los paquetes creados ***import***

- Antes de declarar nuestra clase ***import paquete.clase***

```
import java.util.StringTokenizer;  
class MiClase  
{  
    ...  
    StringTokenizer str = new StringTokenizer();  
    ...  
}
```

- Si deseamos utilizar más clases dentro del paquete

import paquete.*;

■ Java API

□ **java.lang:** Paquete principal. Contiene todas las clases básicas del lenguaje:

- Object: clases base de JFC (*Java Foundation Classes*)
- Thread: clase necesaria para la programación concurrente
- Exception: clase base para el control de errores mediante excepciones
- Clases Contenedoras: Integer, Carácter, Float,
- Manejo de caracteres: String, StringBuffer,

■ Java API (II)

- **java.applet:** Para crear nuestros applets (ver módulo Applets)
- **java.awt:** Contiene las clases necesarias para realizar un interface gráfico de usuario en nuestra aplicación (etiquetas, cuadros de texto, listas desplegadas, botones, ...)
- **java.io:** Contiene las clases necesarias para trabajar con entrada/salida, tanto estándar como sobre ficheros.
- • **java.net**
- Contiene las clases necesarias para trabajar con aplicaciones que accedan a redes

■ Java API (III)

- **java.net:** Contiene las clases necesarias para trabajar con aplicaciones que accedan a redes TCP/IP
- **java.util:** Contiene clases que definen estructuras de datos complejas (tablas hash, arrays, conjuntos) así como otras clases que permiten acceder a recursos del sistema.
- **java.swing:** Contiene las clases necesarias para el diseño de interfaces gráficos de usuario mejorando las clases de java.awt

■ Inclusión de paquetes: Análisis

- ¿Cómo analiza java la inclusión de paquetes?

```
import java.applet.Applet;  
import java.util.*;
```

- Primero buscará en los paquetes cualificados
 - Aquellos que se indica el nombre de la clase dentro del paquete:
import java.applet.Applet;
- Seguidamente en el paquete de la clase (*package*)
- Por último en los importados con *

■ Paquetes: Consideraciones

- Debemos agrupar las clases de los paquetes por funcionalidad
- El paquete 'no es más' que una ubicación física donde almacenar nuestros .class (***classpath***)
- Visibilidad
 - Una clase pública → Total accesibilidad.
 - protected → Sólo clases del mismo paquete.
- El nombre del paquete es un identificador
 - Respetar las reglas para su creación [a..z][A..Z]...

■ Paquetes: Pasos a seguir

1. Definir la ruta donde ubicaremos las clases
 - **set classpath=%classpath%;c:\java\misclases**

2. Definir la clase como pública

```
public class MiClase
{
    ...
}
```

3. Indicar a que paquete pertenecemos

```
package paquetel;
public class MiClase
{
    ...
}
```

4. Incluir las clases necesarias

```
import paquete2.MiOtraClase;
```

■ Paquetes: Ojo Ambigüedad

- Dos clases con el mismo nombre en distintos paquetes
- ¿Qué ocurre cuando queremos usar las dos?
- Ejemplo: *List*
 - `java.util`: interface de comportamiento de una lista
 - `java.awt`: clase control gráfico de tipo lista.
- Solución:
 - En la declaración de la variable se indica la clase a la que pertenece
 - `java.util.list lista;`
 - `java.awt.list lista;`



Conclusiones

1. Programación Orientada a Objetos
2. Introducción y Sintaxis Java
3. Sentencias Control Flujo
4. POO en Java
5. Relaciones entre Objetos
6. Polimorfismo, abstracción e interfaces
7. **Excepciones**
8. Conceptos avanzados

- Excepciones
 - Gestión excepciones
 - Jerarquía
 - Captura
 - Creación
 - Lanzar excepciones
- Paquetes
 - Organización
 - Creación e inclusión
 - JAVA API
 - Consideraciones