

# Parte I: Programación en un lenguaje orientado a objetos

---

1. Introducción a los lenguajes de programación

2. Datos y expresiones

3. Estructuras algorítmicas

4. Datos compuestos

***5. Tratamiento de errores***

- Excepciones. Bloques de tratamiento excepciones. La cláusula finally. Patrones de tratamiento de excepciones. Jerarquía de las excepciones. Lanzar excepciones. Usar nuestras propias excepciones

6. Entrada/salida

7. Herencia y Polimorfismo

# Excepciones

---

Son un mecanismo especial para ***gestionar errores***

- Permiten separar el tratamiento de errores del código normal
- Evitan que haya errores que pasen inadvertidos
- Permiten propagar de forma automática los errores desde los métodos más internos a los más externos
- Permiten agrupar en un lugar común el tratamiento de errores que ocurren en varios lugares del programa
- En Java son clases especiales

Las excepciones se ***lanzan*** para indicar que ha ocurrido un error:

- automáticamente, cuando el sistema detecta un error
- explícitamente cuando el programador lo establezca

Están presentes en los lenguajes más modernos

# Propagación de excepciones



fuentes: <http://free-illustrations.gatag.net/>

# Objetivos de las excepciones

---

Los errores en un método *nunca deben pasar inadvertidos*

Los *errores previsibles*:

- Deben ser *detectados lo antes posible*
- Deben ser *notificados* al método llamante (y quizá también al usuario)
- Su efecto debe ser *corregido* por la aplicación (siempre que sea posible)

Los *errores imprevistos*

- es preferible que *finalicen* la aplicación (con un mensaje que permita su *diagnosis*),
- a que pasen inadvertidos causando un mal funcionamiento del sistema de difícil diagnóstico

# Conceptos asociados a las excepciones

---

## **Lanzar**

- La excepción se lanza para avisar de que hay un error
  - automáticamente
  - o explícitamente con la instrucción `throw`

## **Propagar**

- La excepción se propaga de un bloque al siguiente hasta se trata

## **Tratar**

- Ejecutar las instrucciones de un manejador de excepción
  - para resolver la situación de error

## **Manejador**

- Instrucciones que se escriben para resolver un error

# Ejemplo de lanzamiento automático: División por cero

```
import fundamentos.*;
public class DivisionPorCero {

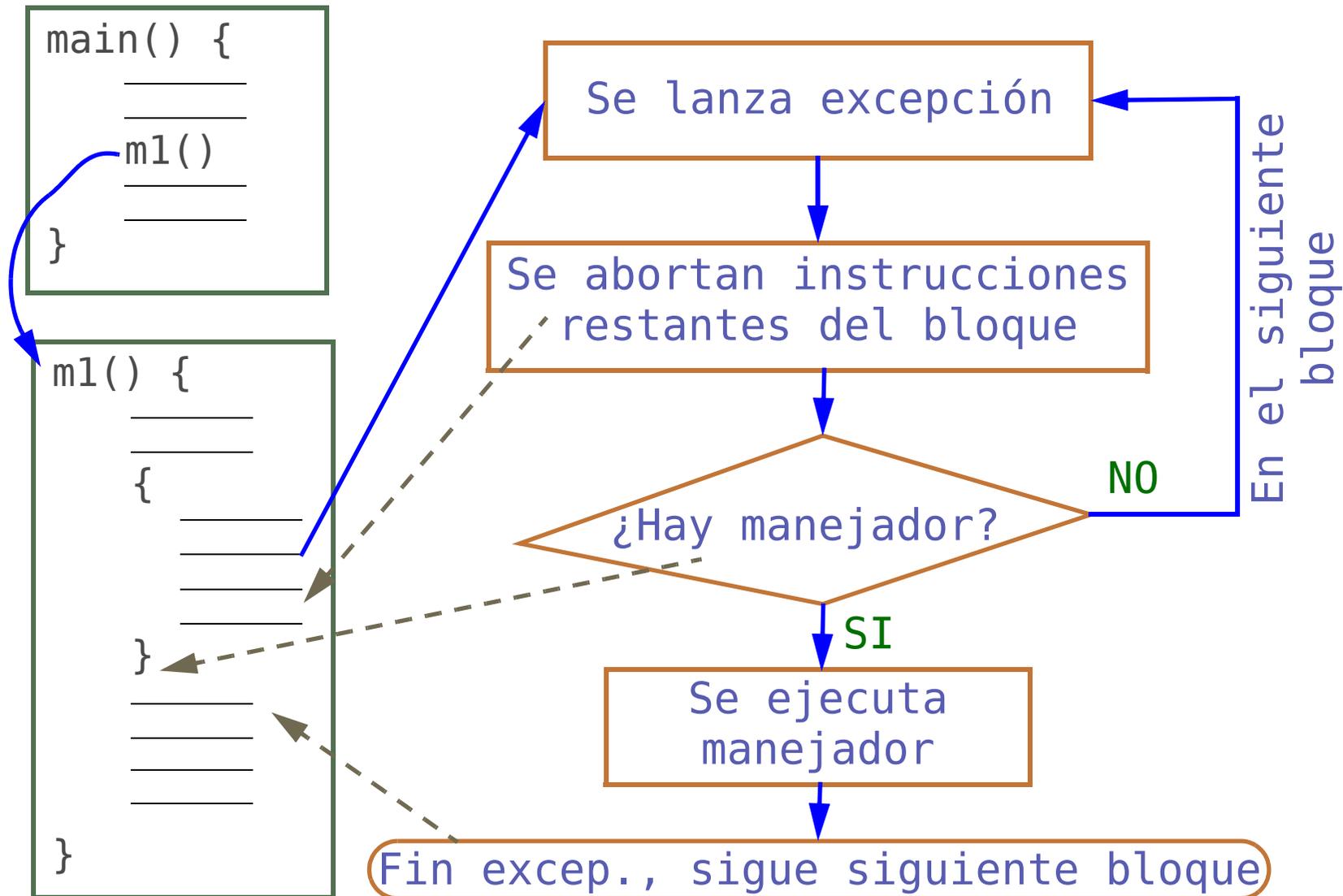
    public static void main(String[] args)
    {
        int i, j, div;
        Lectura leer = new Lectura("Enteros");
        leer.creaEntrada("i",0);
        leer.creaEntrada("j",0);
        leer.espera("introduce datos");
        i=leer.leeInt("i");
        j=leer.leeInt("j");
        System.out.println("Divide...");
        div = i/j;
        System.out.println("i/j="+div);
    } // fin main

} // fin DivisionPorCero
```

cuando **j** vale 0 se lanza  
la excepción  
**ArithmeticException**

cuando se lanza la excepción  
esta línea no se ejecuta

# Propagación de excepciones: en detalle



# Propagación de excepciones

---

Una línea de código **lanza** una excepción

El bloque que contiene esa línea de código se aborta en ese punto

Si el bloque **trata** esa excepción (es decir, si tiene un **manejador** para ella), el manejador se ejecuta

- la “vida” de la excepción finaliza en este punto

Si no tiene manejador, la excepción se **propaga** al bloque superior

- que, a su vez, podrá tratar o dejar pasar la excepción

Si la excepción alcanza el bloque principal (**main**) y éste tampoco trata la excepción, el programa finaliza con un mensaje de error

# Ejemplo de propagación de excepciones

---

```
private static int divide(int a, int b) {
    System.out.println("divide: antes de dividir");
    int div = a/b;
    System.out.println("divide: después de dividir");
    return div;
}

private static void intermedio() {
    System.out.println("intermedio: antes de divide");
    int div = divide(2,0);
    System.out.println("intermedio: resultado:" +div);
}

public static void main(String[] args) {
    System.out.println("main: antes de intermedio");
    intermedio();
    System.out.println("main: después de intermedio");
}
```

# Ejemplo de propagación de excepciones

---

Si hay división por cero la salida generada será:

```
main: antes de intermedio  
intermedio: antes de divide  
divide: antes de dividir
```

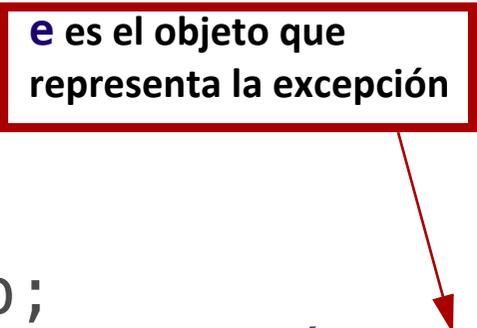
```
java.lang.ArithmeticException: / by zero  
at Propaga.divide(Propaga.java:13)  
at Propaga.intermedio(Propaga.java:20)  
at Propaga.main(Propaga.java:26)
```

# Bloques de tratamiento de excepciones

---

La forma general de escribir un bloque en el que se tratan excepciones es:

```
try {  
    instrucciones;  
} catch (ClaseExcepción1 e) {  
    instrucciones de tratamiento;  
} catch (ClaseExcepción2 | ClaseExcepción3 e) {  
    instrucciones de tratamiento;  
}
```



Los “catch” se evalúan por orden:

- una excepción se trata en el primer “catch” para esa excepción o para una de sus superclases

Desde Java 7 se permiten múltiples excepciones en un “catch”

# Ejemplo: propagación con bloque try-catch

---

En el ejemplo "propagación de excepciones" anterior, añadimos un bloque `try-catch` al método intermedio:

```
private static void intermedio() {
    try {
        System.out.println("intermedio: antes de " +
                           "divide");

        int div=divide(2,0);
        System.out.println("intermedio: resultado:" +
                           div);
    } catch (ArithmeticException e) {
        System.out.println("intermedio: tratado error " + e);
    }
}
```

# Ejemplo: propagación con bloque try-catch

---

La salida por consola que obtenemos ahora es:

```
main: antes de intermedio
intermedio: antes de divide
divide: antes de dividir
intermedio: cazada ArithmeticException: / by zero
main: después de intermedio
```

- en este caso la excepción es tratada, por lo que
  - el programa NO finaliza de forma abrupta
  - NO aparece un mensaje del sistema indicando que se ha producido una excepción

# Tratamiento específico

---

Tratamiento *únicamente* de la excepción `ArithmeticException`

```
try {  
    ...;  
} catch (ArithmeticException e) {  
    ...;  
}
```

Es posible poner un *tratamiento común* para cualquier excepción

```
try {  
    ...;  
} catch (Exception e) {  
    ...;  
}
```

- es cómodo pero *no es recomendable en general*, ya que puede ocurrir un tratamiento inadecuado para una excepción no prevista
- *sí es recomendable* para el `main`, con objeto de poner un mensaje de error apropiado antes de finalizar la aplicación

# La cláusula `finally`

---

Permite crear un bloque de código que se ejecuta siempre después del bloque `try-catch`

- haya habido excepción o no,
- incluso si se sale a causa de `return`, `break` o `continue`

```
try {  
    operaciones;  
} catch (ClaseExcepción1 e) {  
    tratamiento de la excepción;  
} catch (ClaseExcepción2 e) {  
    tratamiento de la excepción;  
} finally {  
    ejecuta siempre;  
}
```

- la cláusula `finally` es opcional

# Patrones de tratamiento de excepciones

---

Según la gravedad del error:

- **leve**: se notifica el error, pero la aplicación continúa
- **grave**: se notifica el error y se finaliza una parte de la aplicación, o la aplicación completa
- **recuperable**: se reintentará la operación

# Patrones de tratamiento de excepciones

---

Esquema de tratamiento de un *error leve*

```
try {  
    instrucciones  
} catch (ClaseExcepción e) {  
    notificación del error leve  
}
```

# Patrones de tratamiento de excepciones

---

Esquema de tratamiento de un *error grave*

```
try {  
    instrucciones  
} catch (ClaseExcepción e) {  
    notificación del error grave  
    System.exit(-1); // finaliza la aplicación  
}
```

En otras ocasiones se finaliza sólo el método (con `return`), o se lanza otra excepción (con `throw`)

# Patrones de tratamiento de excepciones

---

Esquema de tratamiento de *error recuperable*

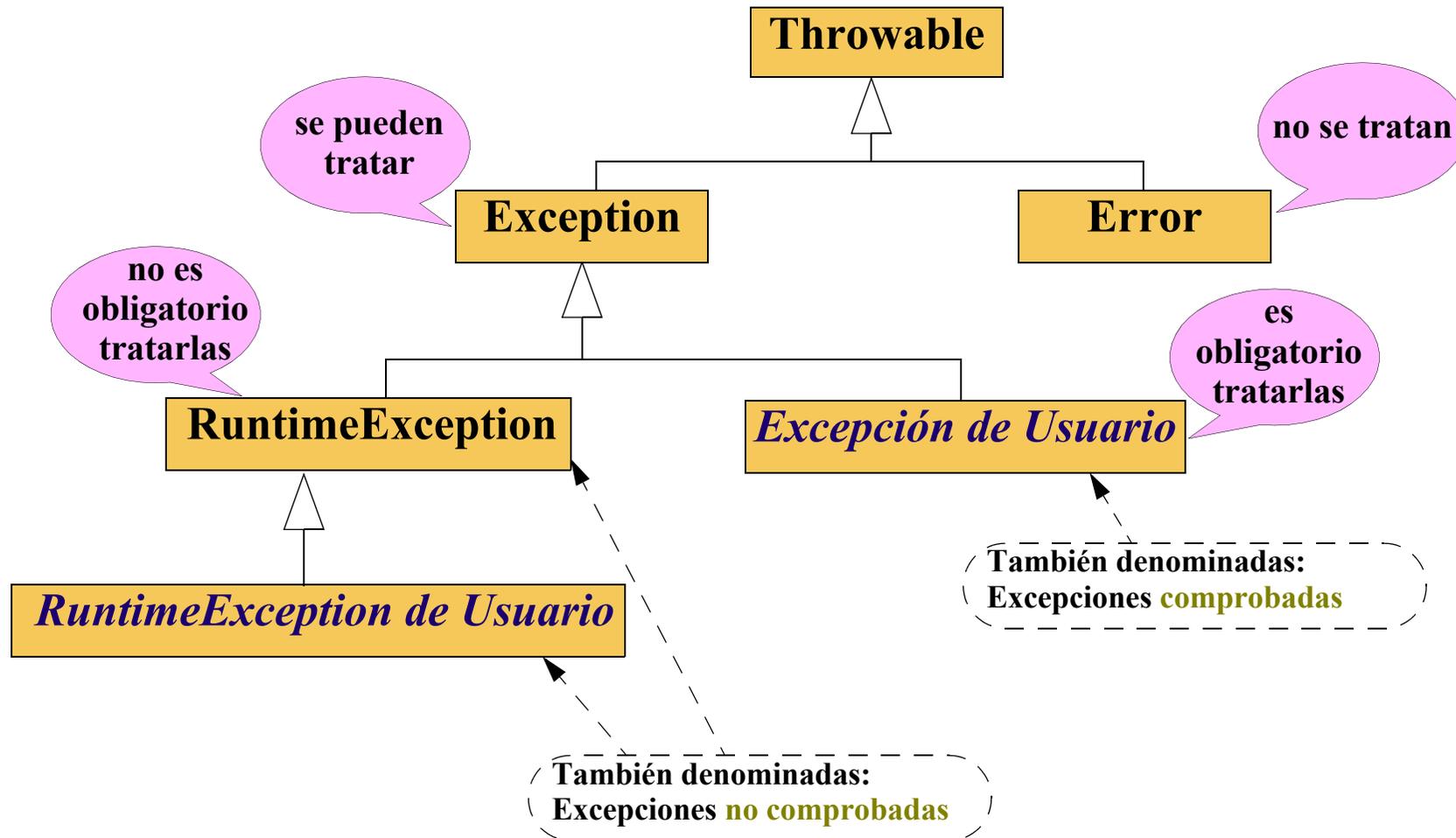
```
boolean correcto = false
do {
    try {
        instrucciones a reintentar
        correcto = true
    } catch (ClaseExcepción e) {
        tratamiento
    }
} while (!correcto);
```

# Ejemplo de error recuperable: lee dos notas

---

```
double nota1, nota2;
boolean notasCorrectas = false;
Lectura lec = new Lectura("Lee notas");
lec.creaEntrada("Nota parcial 1",5.0);
lec.creaEntrada("Nota parcial 2",5.0);
do {
    lec.esperaYCierra("Introduce notas");
    try {
        nota1=lec.leeDouble("Nota parcial 1");
        nota2=lec.leeDouble("Nota parcial 2");
        notasCorrectas = true; // sale del bucle
    } catch (NumberFormatException e) {
        // no muestra mensaje de error porque ya
        // lo hace leeDouble
    }
} while (!notasCorrectas);
```

# Jerarquía de las excepciones



# Algunas excepciones RuntimeException

---

También se denominan excepciones ***no comprobadas***

<code>ArithmeticException</code>	Error aritmético (x/0, ...)
<code>ArrayIndexOutOfBoundsException</code>	Índice de array fuera de límites (<0 o >=length)
<code>ClassCastException</code>	Intento de convertir a una clase incorrecta
<code>IndexOutOfBoundsException</code>	Índice fuera de límites (p.e., en un ArrayList)
<code>NegativeArraySizeException</code>	Tamaño de array negativo
<code>NullPointerException</code>	Uso de una referencia nula
<code>NumberFormatException</code>	Formato de número incorrecto
<code>StringIndexOutOfBounds</code>	Índice usado en un String está fuera de límites

# Lanzar excepciones

---

Se lanzan con la palabra reservada **throw**:

```
throw new ClaseExcepción();
```

En ocasiones puede ser más conveniente usar el constructor con un string como parámetro

```
throw new ClaseExcepción("mensaje");
```

- que sirve para dar información adicional sobre la causa de la excepción

Ejemplo:

```
if (clave==null) {  
    throw new NullPointerException("clave es nula");  
}
```

# Lanzar la misma excepción

---

En algunas ocasiones un manejador puede volver a lanzar la misma excepción:

```
catch (ClaseExcepción e) {  
    parte del tratamiento de la excepción;  
    throw e;  
}
```

- puede ser útil cuando se desea realizar en el manejador parte del tratamiento de la excepción
  - y dejar que el resto del tratamiento le haga el método superior

# Usar nuestras propias excepciones

---

El programador puede crear sus propias excepciones y utilizarlas para indicar errores:

```
public class MiExcepción extends Exception {}
```

Las excepciones creadas por el programador que extienden a la clase `Exception`

- son ***excepciones comprobadas***

Un método donde se lanza una excepción comprobada, deberá:

- tratarla con un bloque `try-catch`
- o declarar en su cabecera que la lanza, con una cláusula `throws`

# Sintaxis de la cláusula throws

---

```
public tipo nombreMétodo(parámetros)
  throws ClaseExcepción1, ClaseExcepción2
{
  declaraciones;
  instrucciones; // lanzan las excepciones
}
```

# Ejemplo de excepción propia

---

Clase que define la excepción

```
public class NoQuieroTrabajar extends Exception {}
```

# Ejemplo de excepción propia

---

Operación que lanza la excepción

```
public class Operador
{
    public String trabaja(String diaSemana)
        throws NoQuieroTrabajar
    {
        if (diaSemana.equals("Miercoles")) {
            return "OK. voy a trabajar";
        } else {
            // si no es miércoles
            throw new NoQuieroTrabajar();
        }
    }
}
```

# Ejemplo de excepción propia

---

Operación que invoca a `trabaja()`, y no la trata

```
public void mandaPepe() throws NoQuieroTrabajar
{
    pepe.trabaja("viernes");
}
```

# Ejemplo de excepción propia

---

Operación que invoca a `trabaja()` y la trata

```
public void mandaJuan()
{
    try {
        juan.trabaja("sábado");
    } catch (NoQuieroTrabajar e) {
        System.out.println
            ("Juan no quiere trabajar");
    }
}
```