

Programación Orientada a Objetos

Java: Excepciones

Eduardo Mosqueira Rey



LIDIA

**Laboratorio de Investigación y
desarrollo en Inteligencia Artificial**



**Departamento de Computación
Universidade da Coruña, España**



Índice

- 1. Introducción**
- 2. Clases de excepciones**
- 3. Excepciones personalizadas**
- 4. La construcción try – catch**
- 5. Aserciones**
- 6. Conclusiones finales**



Excepciones

Introducción

- **Solución tradicional**
 - La solución tradicional consistía en que los métodos devolvieran un valor en el que indicaran si en su ejecución se había producido alguna incidencia que pudiera dar lugar a un error en el futuro.
 - Por ejemplo la función `fopen` del C se utiliza para abrir un fichero en el disco. Si este fichero no puede ser abierto la función devuelve el valor `null`.
- **Desventajas de la solución tradicional**
 - El encargado de llamar a la función debe acordarse de recoger el valor de retorno.
 - El código puede convertirse en una sucesión de comprobaciones de situaciones erróneas.
 - El encargado de llamar a la función puede no saber tratar el error y necesite pasárselo a métodos de niveles superiores.



Excepciones

Introducción

- **Ventajas de las excepciones**
 - Si no se quiere las excepciones no pueden ser obviadas
 - El lenguaje provee de construcciones del tipo try - catch - finally que facilitan la escritura del código en presencia de excepciones y evita tener que incluir sentencias condicionales cada vez que se llama a un método que puede generar una excepción
 - Los métodos que no sepan cómo tratar una excepción pueden pasarla a niveles superiores a través de la cláusula throws



Excepciones

Introducción

- **Piscina sencilla**

```
class Piscina
{
    private int nivel;
    public final int MAX_NIVEL;

    public Piscina(int max)
    {
        if (max<0) max=0;
        MAX_NIVEL=max;
    }

    public int getNivel()
    { return nivel; }

    public void vaciar(int cantidad)
    { nivel=nivel-cantidad; }

    public void llenar(int cantidad)
    { nivel=nivel+cantidad; }
}
```



Excepciones

Introducción

- **Piscina con excepciones**

```
class Piscina
{
    private int nivel;
    public final int MAX_NIVEL;

    public Piscina(int max)
    {
        if (max<0) max=0;
        MAX_NIVEL=max;
    }

    public int getNivel()
    { return nivel; }

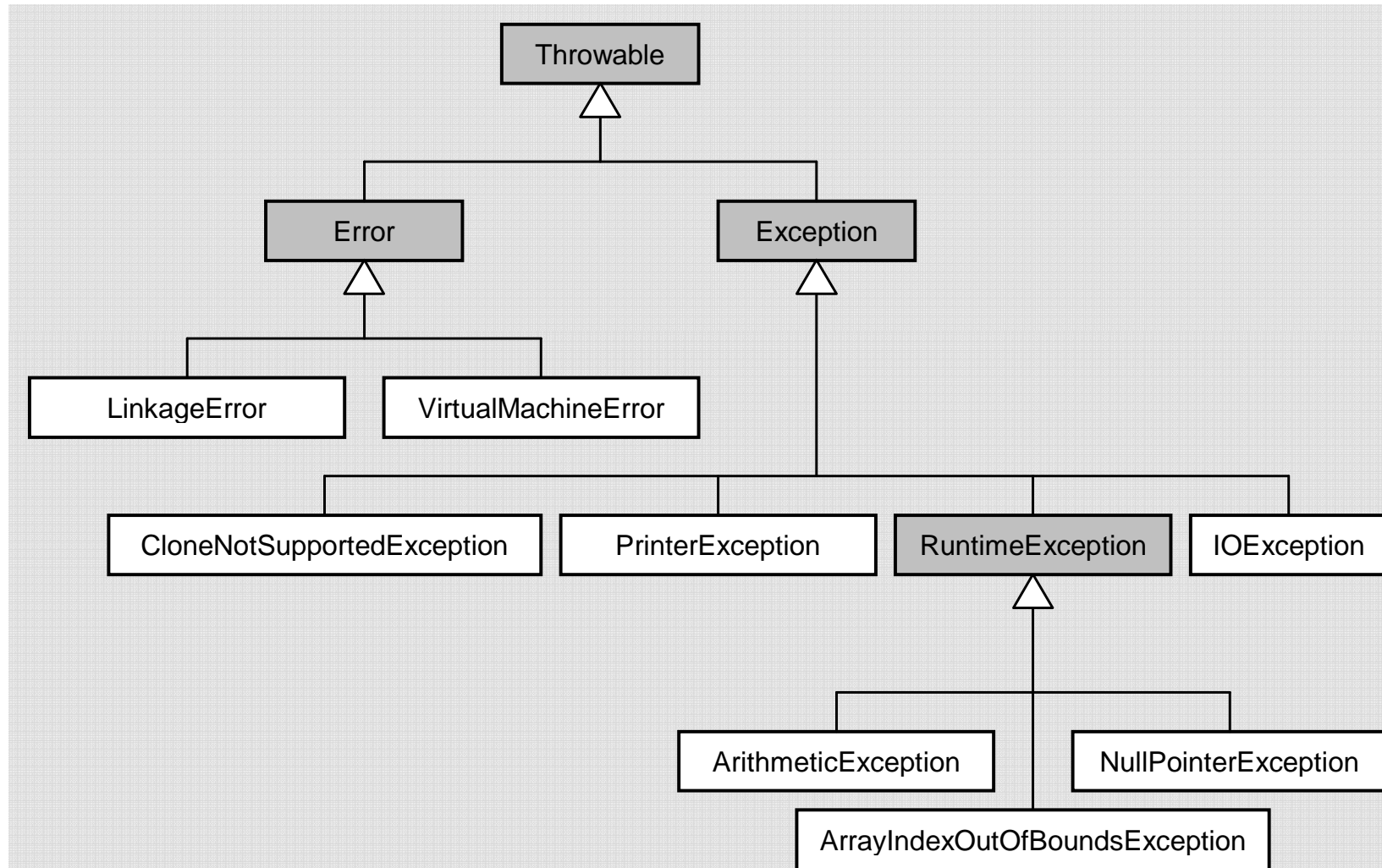
    public void vaciar(int cantidad) throws Exception
    {
        if (nivel-cantidad < 0)
            throw new Exception();
        else nivel=nivel-cantidad;
    }

    public void llenar(int cantidad) throws Exception
    {
        if (nivel+cantidad > MAX_NIVEL)
            throw new Exception();
        else nivel=nivel+cantidad;
    }
}
```



Excepciones

Clases de Excepciones





Excepciones

Clases de Excepciones

- **Características generales**
 - Las clases predefinidas están en el paquete `java.lang`
 - Las clases personalizadas pueden estar en cualquier paquete
- **Throwable**
 - Describe la funcionalidad básica de todo aquello que se puede lanzar en forma de excepción.
 - Dos constructores: uno sin parámetros y otro en el que se le puede incluir una cadena de texto que describa el error producido
- **Error**
 - Destinada a representar problemas graves o condiciones anormales que no deberían ocurrir normalmente por lo que no es necesario que las aplicaciones se ocupen de gestionarlos



Excepciones

Clases de Excepciones

- **Exception**
 - Es la clase base de aquellas excepciones que puede lanzar un programa, por lo que es la clase que más interesa al programador.
 - Tanto Exception como sus subclases (sin incluir RuntimeException) son excepciones comprobadas, en el sentido de que el compilador comprueba que si se lanza una excepción en un método esta debe ser capturada por el propio método o incluida en su cláusula throws.
- **RuntimeException**
 - Junto con sus subclases representan excepciones en tiempo de ejecución que no necesitan ser capturadas obligatoriamente, por lo tanto se trata de excepciones no comprobadas.
 - Las RuntimeException pueden ocurrir en cualquier parte de un programa y, normalmente, de forma muy numerosa. Por ese motivo el coste de comprobar obligatoriamente si ha ocurrido una RuntimeException es mayor que el beneficio que se produce por dicha captura obligatoria.
 - De esta forma Java permite que la captura de estas excepciones sea algo opcional.



Excepciones

Excepciones personalizadas

```
class PiscinaNivelException extends Exception
{
    int nivel;
    public PiscinaNivelException (String descripcion, int valor)
    {
        super (descripcion);
        nivel=valor;
    }
}
```

```
class Piscina
{
    private int nivel;
    public final int MAX_NIVEL;

    public Piscina(int max)
    {
        if (max<0) max=0;
        MAX_NIVEL=max;
    }

    public int getNivel()
    { return nivel; }

    public void vaciar(int cantidad) throws PiscinaNivelException
    {
        if (nivel-cantidad < 0)
            throw new PiscinaNivelException("Vaciado excesivo", nivel-cantidad);
        else nivel=nivel-cantidad;
    }

    public void llenar(int cantidad) throws PiscinaNivelException
    {
        if (nivel+cantidad > MAX_NIVEL)
            throw new PiscinaNivelException("Llenado excesivo", nivel+cantidad);
        else nivel=nivel+cantidad;
    }
}
```



Excepciones

La construcción try - catch

- Captura de excepciones con try - catch

```
class PiscinaCliente
{
    public static void operacionesPiscina(Piscina P)
    {
        try
        {
            for (int i=1;i<=3;i++)
            {
                P.llenar((int)(Math.random()*100));
                System.out.println ("llenado..." + P.getNivel());
                P.vaciar((int)(Math.random()*100));
                System.out.println ("vaciado..." + P.getNivel());
            }
        }
        catch (PiscinaNivelException e)
        {
            System.out.println(e.toString() + ' ' + e.nivel);
        }
        System.out.println("El nivel de la piscina es..." + P.getNivel());
    }

    public static void main (String [] args)
    {
        Piscina P = new Piscina(100);
        operacionesPiscina(P);
    }
}
```



Excepciones

La construcción try - catch

- **Múltiples sentencias catch**

```
try
{ ... }
catch (Excepcion_1 identificador_1)
{ ... }
catch (Excepcion_2 identificador_2)
{ ... }
finally
{ ... }
```

- **Ejemplo de mala utilización**

```
try
{
    throw PiscinaNivelException;
}
catch (Exception e)
{
    // Captura Exception y PiscinaNivelException
}
catch (PiscinaNivelException e)
{
    // Este código nunca se ejecuta
}
```



Excepciones

La construcción try - catch

- throws como alternativa al try - catch

```
class PiscinaCliente
{
    public static void operacionesPiscina(Piscina P) throws PiscinaNivelException
    { for (int i=1;i<=3;i++)
      {
        P.llenar((int)(Math.random()*100));
        System.out.println ("llenado..." + P.getNivel());
        P.vaciar((int)(Math.random()*100));
        System.out.println ("vaciado..." + P.getNivel());
      }
      System.out.println("El nivel de la piscina es..." + P.getNivel());
    }

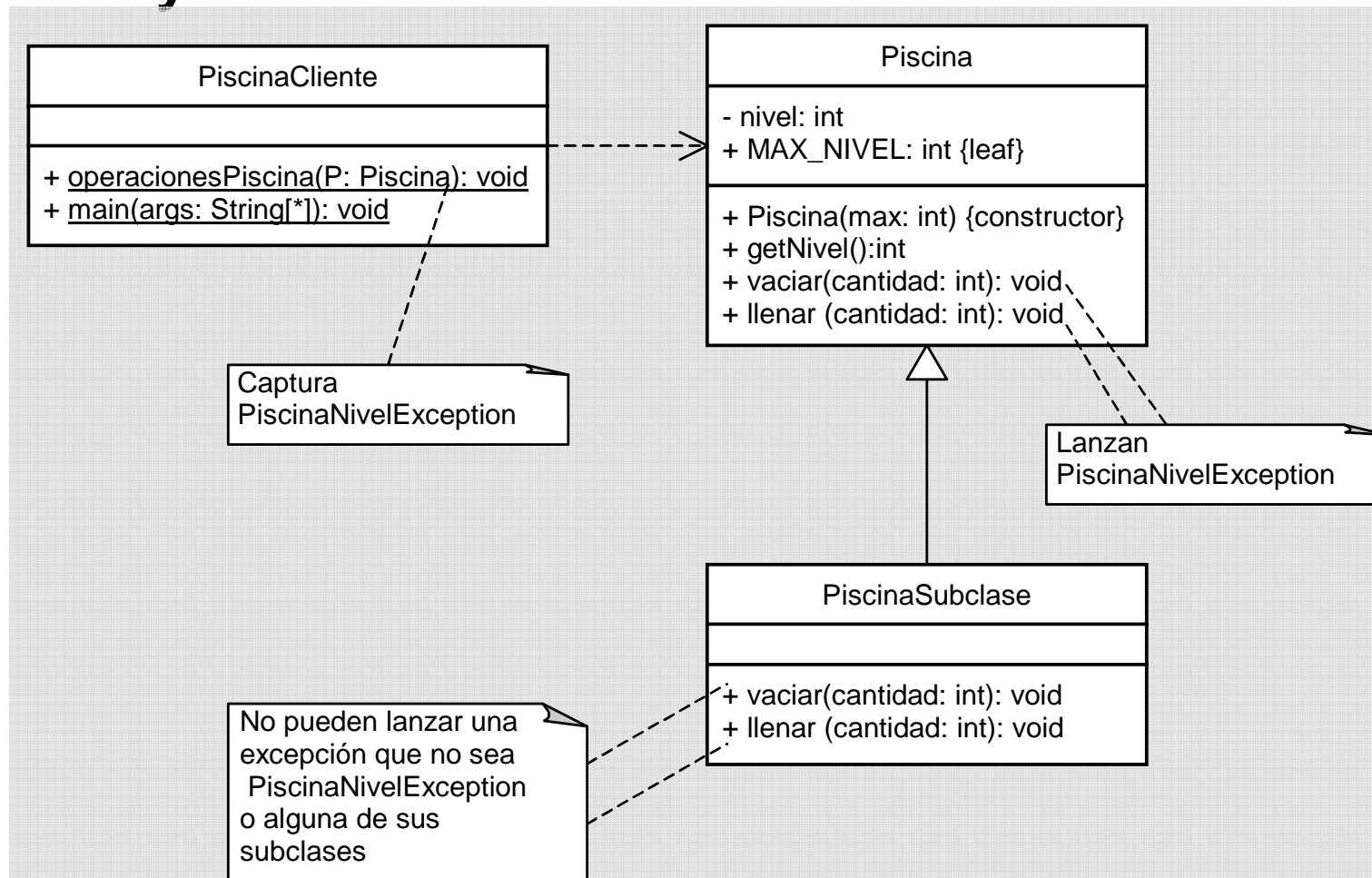
    public static void main (String [] args)
    { Piscina P = new Piscina(100);
      try
      { operacionesPiscina(P); }
      catch (PiscinaNivelException e)
      { System.out.println(e.toString() + ' ' + e.nivel); }
    }
}
```



Excepciones

La construcción try - catch

- throws y la sobrescritura





Excepciones

Aserciones

- **Aserciones**
 - Comprobaciones introducidas en el código fuente para verificar precondiciones, invariantes de bucle, postcondiciones, etc.
- **Formato**
 - `assert expresión_booleana;`
 - `assert expresión_booleana : expresión_2;`
- **Funcionamiento**
 - La expresión booleana se evalúa y en caso de ser falsa se lanza la excepción `AssertionException`
 - `expresión_2` es una expresión que devuelve un valor (no puede devolver `void`). Este valor es convertido a `String` y pasado al constructor de `AssertionException` para detallar más en profundidad la aserción



Excepciones

Aserciones

- **Ventajas**
 - Introducir aserciones en nuestro código facilita la detección de errores
 - Además las aserciones pueden deshabilitarse para que, una vez finalizado el periodo de pruebas, no afecten al rendimiento del sistema
- **Recomendación**
 - En las precondiciones de métodos se suele recomendar no usar aserciones sino excepciones (`NullPointerException`, `IllegalArgumentException`) porque suelen dar más información que un genérico `AssertException`
- **Utilización**
 - “assert” es una palabra clave desde la versión 1.4 de Java
 - Por lo que al compilar con el JDK 1.4 es necesario hacer lo siguiente:
`javac -source 1.4 fichero.java` (con el JDK 1.5 no es necesario)
 - Al interpretar código java con aserciones hay dos posibilidades
 - `java -ea fichero` (`ea` \equiv enable assertions) activa las aserciones
 - `java -da fichero` (`da` \equiv disable assertions) desactiva las aserciones (es la opción por defecto)



Excepciones

Aserciones

- **Ejemplo**

```
switch(palo)
{
    case Palo.ESPADAS:
        ...
        break;

    case Palo.COPAS:
        ...
        break;

    case Palo.BASTOS:
        ...
        break;

    case Palo.OROS:
        ...
        break;

    default:
        assert false : palo;
}
```

- **Para más información**

- <http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html>



Excepciones

Conclusiones finales

- Las excepciones comprobadas tiene la característica de que obligas al programador a tenerlas en cuenta, pero lo que generalmente se hacía era capturarlas con cláusulas catch vacías => cuando suceda una excepción el error puede pasar inadvertido.
- Mas grave aún era ver construcciones que declaran la excepción en la cláusula throws de todos los métodos de la pila de ejecución (incluido el main).
- Por dicho motivo la gente ha preferido usar las RuntimeException y dejar la posibilidad de capturar o no la excepción al programador (menos seguro pero menos molesto)
- La norma general es que se restrinja el uso de excepciones a aquellas situaciones erróneas o inesperadas, no utilizarlas para capturar situaciones previsibles (llegamos al final de un fichero de entrada).