Estructura de las Aplicaciones Orientadas a Objetos

# Excepciones

*Programación Orientada a Objetos*
*Facultad de Informática*

**Juan Pavón Mestras**
**Dep. Ingeniería del Software e Inteligencia Artificial**
**Universidad Complutense Madrid**

*Basado en el curso Objects First with Java - A Practical Introduction using BlueJ, © David J. Barnes, Michael Kölling*

## Conceptos

- Programación a la defensiva
  - Anticiparse a lo que podría ir mal
- Lanzamiento y tratamiento de excepciones
- Aserciones

## Causas de situaciones de error

- Implementación incorrecta
  - No se ajusta a las especificaciones
- Petición de objeto inapropiada
  - Índice inválido
  - Referencia nula
- Estado de objeto inapropiado o incosistente
  - P.ej. debido a una extensión de la clase



## No siempre se trata de errores de programación

- Hay errores que vienen del entorno:
  - URL incorrecta
  - Interrupción de las comunicaciones de red
- El procesamiento de ficheros suele ocasionar errores:
  - Ficheros que no existen
  - Permisos inapropiados

## Explorando los errores

- Explora posibles errores en los proyectos de la agenda (*address-book)*
- Dos aspectos:
  - Informe de errores
  - Tratamiento de errores



## Programación a la defensiva

- Interacción cliente-servidor
  - ¿Tiene que asumir un servidor que todos los clientes se comportarán adecuadamente?
  - ¿O debería asumir que algunos clientes pueden tener un comportamiento hostil?
- Esto implica diferencias significativas en la implementación

## Aspectos a tener en cuenta

- ¿Cuántas comprobaciones tiene que hacer un servidor en las llamadas a sus métodos?
- ¿Cómo informar de los errores?
- ¿Cómo puede anticiparse un cliente a los fallos?
- ¿Cómo debería tratar un cliente un fallo?



## Ejemplo

- Crea un objeto AddressBook
- Intenta eliminar una entrada
- Ocurre un error de ejecución
  - ¿De quién es la culpa?
- La anticipación y la prevención son preferibles a quejarse inútilmente

```
public void removeDetails(String key)
{
   ContactDetails details = book.get(key);
   book.remove(details.getName());
   book.remove(details.getPhone());
   numberOfEntries--;
}
```

## Valores de los argumentos

- Los argumentos son el primer punto de vulnerabilidad de un objeto servidor
  - Los argumentos de un constructor inicializan el estado
  - Los argumentos de los métodos contribuyen a la evolución del comportamiento
- La comprobación de argumentos es una medida defensiva



## Comprobación de la clave

```
public void removeDetails(String key)
{
    if(keyInUse(key)) {
        ContactDetails details = book.get(key);
        book.remove(details.getName());
        book.remove(details.getPhone());
        numberOfEntries--;
    }
}
```

## Informe de errores por el servidor

- ¿Cómo informar de los errores en los argumentos?
  - ¿Al usuario?
    - ¿Hay un usuario humano?
    - ¿Podría resolver el problema?
  - ¿Al objeto cliente?
    - Devuelve un valor de diagnóstico
    - ***Lanza una excepción***



## Devolviendo un diagnóstico (clásico)

```
public boolean removeDetails(String key)
{
    if(keyInUse(key)) {
        ContactDetails details = book.get(key);
        book.remove(details.getName());
        book.remove(details.getPhone());
        numberOfEntries--;
        return true;
    }
    else {
        return false;
    }
}
```

## Problemas del planteamiento clásico

- El cliente comprueba si todo ha ido bien o no según lo que devuelva el método llamado:

```
if ( objeto.función(parámetros) == UN_ERROR ) {
    // tratar error
  }
else {
  // código normal
}
```

  - Mecanismo simple
  - Pero complica el código (cuando hay varias funciones empiezan a anidarse los if-else)
  - Hay errores de ejecución que no pueden capturarse fácilmente
    - Por ejemplo, en C++ al quedarse sin memoria al hacer new o cuando se cae una conexión



## Mecanismo de excepciones

- En lenguajes de POO modernos se usan excepciones
- El código se estructura:
  - En el código del objeto servidor se puede indicar la ocurrencia de una excepción y esto ocasiona la finalización inmediata del método
    - Por ejemplo el incumplimiento de alguna condición necesaria para la ejecución del método
    - También pueden ocurrir excepciones del entorno (no originadas explícitamente por la aplicación)
      - Síncronas: división por cero, acceso fuera de los límites de un array, puntero nulo
      - Asíncronas: algún fallo del sistema, caída de comunicaciones, falta de memoria
  - El código del cliente se puede estructurar en dos partes:
    - Secuencia normal de ejecución
    - Tratamiento de excepciones: qué hacer cuando se sale del flujo normal de ejecución por la ocurrencia de alguna excepción

## Excepciones en Java

- Las excepciones son objetos
  - De la clase *Exception,* que hereda de *Throwable*
- En Java puede haber dos tipos de excepciones:
  - Excepciones que no requieren comprobarse
    - Errores y excepciones de ejecución
    - Clase *RuntimeException*
  - Excepciones que hay que comprobar
    - Todas las demás
    - Heredan de la clase *Exception*
- Cuando ocurre una excepción se dice que se *lanza (throw)*
  ***throw*** *new Excepción();*
- La excepción puede ser capturada para tratarla *(catch)*
  ***catch*** *(Excepción e) { tratamiento(); }*



## Lanzamiento de una excepción

```
/**
 * Look up a name or phone number and return the
 * corresponding contact details.
 * @param key The name or number to be looked up.
 * @return The details corresponding to the key,
 *         or null if there are none matching.
 * @throws NullPointerException if the key is null.
 */
public ContactDetails getDetails(String key)
{
    if(key == null) {
        throw new NullPointerException(
                            "null key in getDetails");
    }
    return book.get(key);
}
```

## Lanzamiento de una excepción

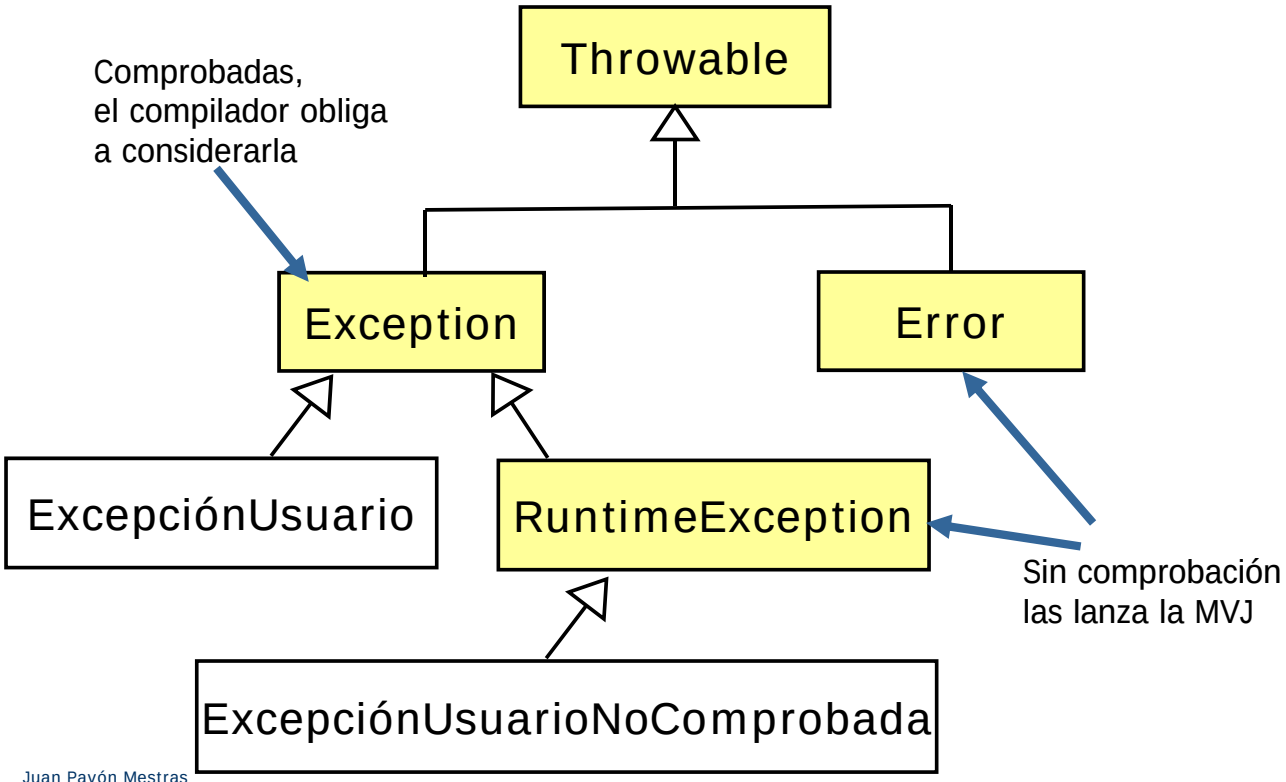
- Se construye un objeto de excepción:
  `new ExceptionType("...");`
- Se lanza el objeto de excepción:
  `throw ...`
- Documentación con Javadoc:
  - `@throws ExceptionType razón`



## Efectos de una excepción

- El método lanzado acaba prematuramente
- No se devuelve ningún valor de retorno
- El control no vuelve al punto de llamada del cliente
  - Con lo cual el cliente no puede despreocuparse
  - Un cliente puede capturar (“catch”) una excepción

## Excepciones en Java






## Categorías de excepción

- Excepciones sin comprobación obligatoria
  - Subclases de `RuntimeException`
  - Son las que puede lanzar la MVJ
    - Normalmente representan una condición fatal del programa
    - No las comprueba el compilador y es difícil saber cuándo y por qué pueden suceder
    - Pero a veces se puede considerar que alguna condición podría ocasionarlas
      - Se suelen tratar en algún nivel superior de forma genérica
- Excepciones con comprobación
  - Subclases de `Exception`
  - Las comprueba el compilador
    - Si no se consideran en el código, el compilador indica un error
    - Estas excepciones son lanzadas por métodos que se usan en el código (por eso las reconoce el compilador)

## Algunas excepciones comunes en Java

- No comprobadas, subclases de *RuntimeException*:
  - NullPointerException
    - Cuando se envía un mensaje a un objeto null
  - ArrayIndexOutOfBoundsException
    - Cuando se accede a un índice ilegal en un array
- Comprobadas:
  - IOException
    - Clase genérica para las excepciones que se producen en operaciones de E/S
  - NoSuchMethodException
    - Cuando no se encuentra un método
  - ClassNotFoundException
    - Cuando una aplicación intenta cargar una clase pero no se encuentra su definición (el fichero .class correspondiente)



## Algunos errores comunes en Java

- NoSuchMethodError
  - La aplicación llama a un método que ya no existe en la definición de la clase
    - Sólo puede ocurrir si la clase cambia su definición en tiempo de ejecución
- NoClassDefFoundError
  - La MVJ intenta cargar una clase pero no se encuentra
    - Suele ocurrir si el CLASSPATH no está bien o si la clase (.class) no está donde se espera
- ClassFormatError
  - La MVJ intenta cargar una clase desde un fichero incorrecto
    - Suele ocurrir cuando el fichero .class está corrupto, o si no es un fichero .class

## Comprobación de argumentos

```
public ContactDetails getDetails(String key)
{
    if(key == null) {
        throw new NullPointerException(
                        "null key in getDetails");
    }
    if(key.trim().length() == 0) {
        throw new IllegalArgumentException(
                    "Empty key passed to getDetails");
    }
    return book.get(key);
}
```



## Prevención de la creación de objetos

```
public ContactDetails(String name, String phone, String address)
{
    if(name == null) {
        name = "";
    }
    if(phone == null) {
        phone = "";
    }
    if(address == null) {
        address = "";
    }

    this.name = name.trim();
    this.phone = phone.trim();
    this.address = address.trim();

    if(this.name.length() == 0 && this.phone.length() == 0) {
        throw new IllegalStateException(
                "Either the name or phone must not be blank.");
    }
}
```

## La cláusula *throws*

- Los métodos que lanzan una excepción controlada deben declararla con la cláusula *throws* :

```
public void saveToFile(String destinationFile)
      throws IOException
```



## Tratamiento de excepciones en el cliente

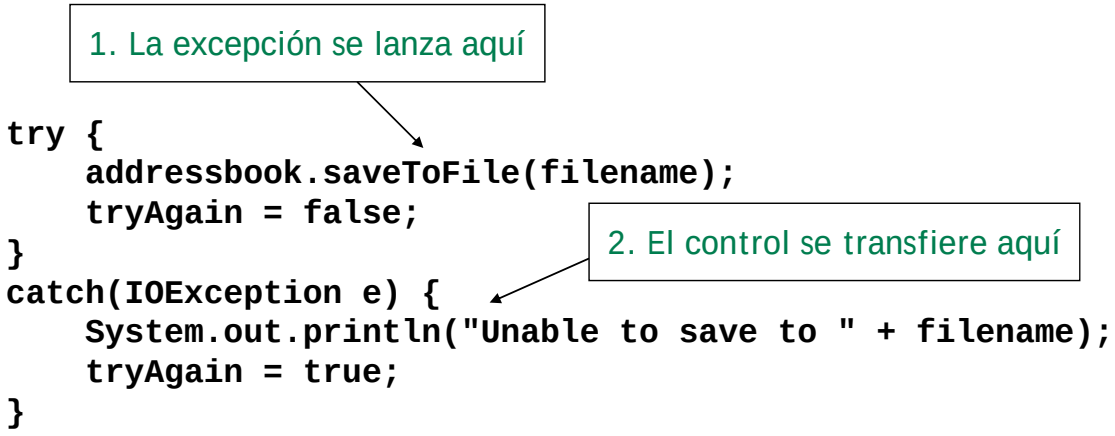
- Hay dos mecanismos para gestionar una excepción:
  - Tratarla en el método que las captura
  - Propagarla al método llamante
    - *Si al final nadie captura la excepción, el programa acaba y se lista la traza de la pila de llamadas*

*Cómo se haga depende del diseño general del sistema*

- Se manejan en un bloque ***try-catch*** (si no, se propagan hacia el llamante):

```
public void unMétodo(){
   try{
        //código donde se puede lanzar la excepción e
   }catch(Exception e){
        //código que gestiona la exception e
   }
}
```

## La sentencia *try*



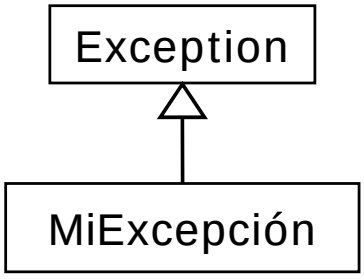

```
try {
    addressbook.saveToFile(filename);
    tryAgain = false;
}
catch(IOException e) {
    System.out.println("Unable to save to " + filename);
    tryAgain = true;
}
```



## Se pueden capturar múltiples excepciones

- El orden de captura de las excepciones es importante
  - Las más genéricas se deberían capturar al final

```
public void unMétodo(){
   try{
     // código donde se puede lanzar la excepción e1
     // código donde se puede lanzar la excepción e2
   }catch(MiExcepción e1){
     // código que gestiona la exception e1
   }catch(Exception e2){
     // código que gestiona la exception e2
   }
}
```



## La cláusula *finally*

- Se ejecuta siempre al final, después del último bloque *catch*
  - Generalmente se utiliza para hacer limpieza
    - Cerrar ficheros, conexiones, etc.

```
public void miMétodo(){
  try{
        // código donde se puede lanzar la excepción e1
        // código donde se puede lanzar la excepción e2
  } catch(MiExcepción e1){
        // código que gestiona la exception e1
  } catch(Exception e2){
        // código que gestiona la exception e2
  } finally{
        //código de limpieza, liberar recursos (close)
  }
}
```



## La cláusula *finally*

- Se ejecuta incluso cuando haya una sentencia de return dentro de las cláusulas try o catch
- Una excepción no capturada o propagada también pasa antes por la cláusula finally

## Gestión genérica de excepciones

- Se puede capturar una excepción genérica para capturar todas las que sean de ese tipo y subtipos
  - Por ejemplo, capturando Exception se pueden capturar todas las excepciones posibles
  - Y capturando Throwable todas las excepciones y errores
    - Lo más genérico es imprimir la traza de la pila de llamadas y así se ve dónde ha ocurrido

```
public void miMétodo(){
   try{
        //código
   }catch(Throwable e){
        System.out.println(e.printStackTrace());
   }
}
```



## Definición de nuevas excepciones

- Excepciones definidas por el usuario
  - Se puede extender RuntimeException para excepciones no comprobadas
  - O Exception para excepciones comprobadas
- La definición de nuevos tipos permite una mejor información sobre la causa de la excepción
  - Puede incluir información de la causa y de recuperación

## Ejemplo de definición de excepción

```
public class NoMatchingDetailsException extends Exception
{
    private String key;

    public NoMatchingDetailsException(String key)
    {
        this.key = key;
    }

    public String getKey()
    {
        return key;
    }

    public String toString()
    {
        return "No details matching '" + key +
               "' were found.";
    }
}
```



## Aserciones

- Usadas para comprobaciones de consistencia interna
  - P.ej. sobre el estado de un objeto
- Normalmente se usan durante el desarrollo y se eliminan en la versión de producto
  - Con una opción del compilador
- Java tiene una sentencia *assert*

## Sentencia de aserción en Java

- Tiene dos formas:
  - **assert** *boolean-expression*
  - **assert** *boolean-expression* **:** *expression*
- La expresión booleana declara algo que debería ser cierto en ese punto
- Se lanza un *AssertionError* si fuera falsa



## Sentencia *assert*

```
public void removeDetails(String key)
   {
       if(key == null){
           throw new IllegalArgumentException("...");
       }
       if(keyInUse(key)) {
           ContactDetails details = book.get(key);
           book.remove(details.getName());
           book.remove(details.getPhone());
           numberOfEntries--;
       }
       assert !keyInUse(key);
       assert consistentSize() :
              "Inconsistent book size in removeDetails";
   }
```

## Guía para aserciones

- No son una alternativa al lanzamiento de excepciones
- Se usan para comprobaciones internas
- Se eliminan para el código de producto final
- No incluyen funcionamiento normal:

```
// Incorrect use:
  assert book.remove(name) != null;
```



## Recuperación de errores

- Los clientes deberían tratar las notificaciones de errores
  - Comprobar los valores de retorno
  - No ignorar las excepciones
- Incluir código para intentar la recuperación
  - Por ejemplo reintentos

## Intento de recuperación

```
// Try to save the address book.
boolean successful = false;
int attempts = 0;
do {
    try {
        addressbook.saveToFile(filename);
        successful = true;
    }
    catch(IOException e) {
        System.out.println("Unable to save to " + filename);
        attempts++;
        if(attempts < MAX_ATTEMPTS) {
            filename = an alternative file name;
        }
    }
} while(!successful && attempts < MAX_ATTEMPTS);
if(!successful) {
    Report the problem and give up;
}
```



## Evitar errores

- Los clientes pueden a veces utilizar métodos de consulta del servidor para evitar errores
  - Clientes más robustos implica que los servidores son más fiables
  - Se pueden usar excepciones no comprobadas
  - Simplifica la lógica de los clientes
- Puede incrementar el acoplamiento cliente-servidor

## Evitar una excepción

```
// Use the correct method to put details
// in the address book.
if(book.keyInUse(details.getName() ||
        book.keyInUse(details.getPhone()) {
    book.changeDetails(details);
}
else {
    book.addDetails(details);
}
```

- El método **addDetails** podría lanzar ahora una excepción no comprobada



## Resumen

- Los errores en tiempo de ejecución ocurren por múltiples motivos
  - Una llamada inapropiada de un cliente a un objeto servidor
  - Un servidor incapaz de completar una petición
  - Un error de programación en el cliente o en el servidor
- La programación a la defensiva anticipa errores, tanto en cliente como en servidor
- Las excepciones permiten la implementación de mecanismos de informe y recuperación de errores

# Ventajas del mecanismo de excepciones

- Separación del tratamiento de errores del resto del código del programa
  - Flujo del programa más sencillo
  - Evita manejos de códigos de error
- Propagación de errores a lo largo de la pila de llamadas a métodos
  - Evitar retornos continuos en caso de error
  - Evitar la necesidad de argumentos adicionales
    - Por ejemplo, el clásico boolean
- Agrupamiento y definición de tipos de errores como clases
  - Jerarquías de excepciones
  - Tratar errores a diferentes niveles de especificidad