Excepciones

- Informática III

Objetivos

- Ver modelos de excepciones en lenguajes de programación
- Uso de excepciones para obtener tolerancia a fallos

Excepciones

- En la práctica, la mayor parte de un sistema, está dedicada a tratar con situaciones anormales, excepcionales o no deseadas; la parte más pequeña corresponde a la propia aplicación: más de 2/3 del código! está dedicada a la detección y manejo de errores (Cristian, 1982)
- Es muy probable que esta porción, contenga errores (*bugs*) ya que, además de su complejidad, rara vez se ejecuta y tampoco se ensaya, documenta ni entiende adecuadamente
- El ejemplo más citado en la literatura es el accidente del cohete Ariane 5 debido a una excepción de software no manejada.

Excepciones: Caso del Ariane 5

Vuelo 501, 4 de Junio de 1996, autodestrucción en H0 +39 :

Costo: 500 millones de dólares

15 años después de Cristian (Cabral y Marques)

```
// The FileWriter must be declared outside of the try block
// and be pointing to something (null is a common choice)
FileWriter file = null;

try
{
    // Open file
    file = new FileWriter("data.txt");

    // Write some data into it
    for (int i=0; i<1024; i++)
        file.write("Here's the data: " + i);
}
catch (FileNotFoundException e)
{
    // Deal with filename problems
}
catch (SecurityException e)
{
    // Deal with problems like wrong permissions
}
catch (IOException e)
{
    // Deal with other (generic?) I/O problems
    // How do I do this???
}
finally
{
    try
    {
        file.close();
    }
    catch (IOException e)
    {
        // What should I do???
    }
}
```

Listing 1.2 – Writing to a file

Excepciones

- Aunque el trabajo de Goodenough (1975) puso los cimientos de la terminología relacionada con las excepciones, aún hoy día, no existe un acuerdo en la definición del concepto de excepción, como así tampoco en cuanto a su utilización
- En realidad, la palabra excepción sugiere “algo que ocurre muy raramente”. Sin embargo, esto se condice sólo parcialmente, con el uso que comúnmente se hace de ellas en la práctica diaria de la programación

Excepciones

- Las excepciones constituyen una forma adicional de pasar información al invocante de un método
- Por tanto, resuelven el problema de los lenguajes de programación que permiten un único valor de retorno
- Pueden utilizarse para distintos propósitos. Goodenough (1975) señala 3:
  - Señalar una avería
  - Clasificar un resultado (Ej.: overflow en suma, EOF), información adicional del resultado para que el invocante pueda interpretarlo adecuadamente
  - Control (Ej. "se han procesado n registros"), el invocante desea que se le notifique que se alcanzó alguna condición
- Por tanto, no necesariamente indican "eventos excepcionales". Ej.: en Java `InterruptedException` se utiliza para sincronizar threads

Excepciones y su uso

- Cuidado con su uso! Debería volverse a las raíces y recordar que las excepciones son **excepcionales**, usarlas para eventos raros (que no ocurren frecuentemente), no para el control del flujo normal. Lamentablemente no está de acuerdo con el uso actual de las excepciones en lenguajes tales como Java.
- En el siguiente ejemplo el 1º método es 750 veces más lento que el 2º (bajo Windows XP y Java 1.4) si se llama con una referencia que no sea del tipo Integer

Ejemplo

```
public static boolean testForInteger1(Object x)
    {
try {
Integer i = (Integer) x;
return true;}
catch (Exception e) {
return false;}}
public static boolean testForInteger2(Object x)
    {return x instanceof Integer;}
```

Manejo de Excepciones

- Un mecanismo de excepciones debe cumplir una serie de requisitos:
  - R1 (**simplicidad**): Sencillo de comprender y utilizar
  - R2 (**discreción**): No afecta la claridad del código ni su comprensión

    R1 y R2 son cruciales en el diseño de sistemas fiables!
  - R3 (**eficiencia**): Introduce sobrecarga en la ejecución sólo cuando se maneja una excepción
  - R4 (**uniformidad**): Provee tratamiento uniforme de las excepciones del entorno y el software de aplicación
  - R5 (**recuperación**): Permite la programación de acciones de recuperación

Mecanismos de Manejo

- Retorno de un valor inusual (C)
- Bifurcación forzada (Assembler)
- `goto` no local
- Excepciones (C++, Java)

Retorno de Valor Inusual (C)

```
if( fopen("archiv.dat", "r") != 0 ){
      /* código de manejo de error */
} else {
      /* código normal */
};
```

☺ R1 ☹ R2 ☹ R3 ☹ R4 ☺ R5

Bifurcación Forzada (Assembler)

```
jsr pc, IMPRIME_SIMB
jmp ERROR_ES
jmp DISPOSITIVO_NO_PREPARADO
# Procesamiento normal
```

☹ R1 ☹ R2 ☺ R3 ☹ R4 ☺ R5

En caso de retorno normal la subrutina que envía un símbolo a un dispositivo, incrementa la dirección de retorno (contador de programa) en dos instrucciones jmp

goto No Local (Salto incondicional global)

```
on error goto err_sub//versión de alto nivel del anterior
...
Procedure Sub_1                    ☹ R1
...                                ☹ R2
Endproc                            ☺ R3
Procedure Sub_2                    ☺ R4
Endproc                            ☺ R5
Procedure err_sub
// tratamiento de errores
Endproc
```

Manejo de excepciones moderno

- Mecanismo estructurado para el tratamiento de las excepciones
- Provisto por el entorno de ejecución (*runtime*)
- Soportado directamente por el lenguaje de programación
- Provee tratamiento uniforme de las excepciones del entorno y del programa
- Mínima sobrecarga
- Soporte de múltiples manejadores de excepciones

Manejo de excepciones moderno

- Lenguajes que tienen incorporado el manejo de excepciones:
  - C++
  - Java
  - Visual Basic
  - Delphi /Pearl/Eiffel/Ada…
  - Todos los lenguajes .NET

Excepciones y su representación

- Síncronas: respuesta inmediata a una operación inapropiada de un fragmento de código
- Asíncronas (**notificación asíncrona o señal**): generadas tiempo después de que ocurra la operación que da lugar a la aparición del error. Puede generarse desde el proceso que ejecutó la operación originalmente, o en otro (contexto de aplicaciones concurrentes). Ej.:ThreadDead

Excepciones síncronas

- Hay dos modelos para declararlas:
  - Una constante $\Rightarrow$ hace falta una declaración explícita (Ada)
  - Un objeto de un tipo particular. No siempre hace falta declararlo explícitamente. (Java)

Dominio de un Manejador de Excepciones

- Dentro de un programa puede haber varios **manejadores** para una misma excepción
- Cada manejador tiene asociado un **dominio** o **ámbito**
- La **granularidad** del dominio determinará qué tan precisamente puede localizarse la fuente de la excepción. No todos los lenguajes permiten alcanzar un nivel de granularidad adecuado.

Dominio de un Manejador de Excepciones

```
Bloque Protegido {//bloque vigilado(guarded)
   // sentencias que pueden generar una
      excepción
}
Manejador para Excepcion Tipo e1{
}
Manejador para Excepcion Tipo en{
}
```

Ámbito de los manejadores en Java

- No todos los bloques tienen manejadores de excepciones
- El ámbito de un manejador se indica explícitamente mediante un bloque vigilado (*guarded*), con la palabra *try*

```
try {
   // instrucciones normales
}
catch (excepcion_type E) {
   // manejador de excepción
}
```

Ámbito de los manejadores en Java

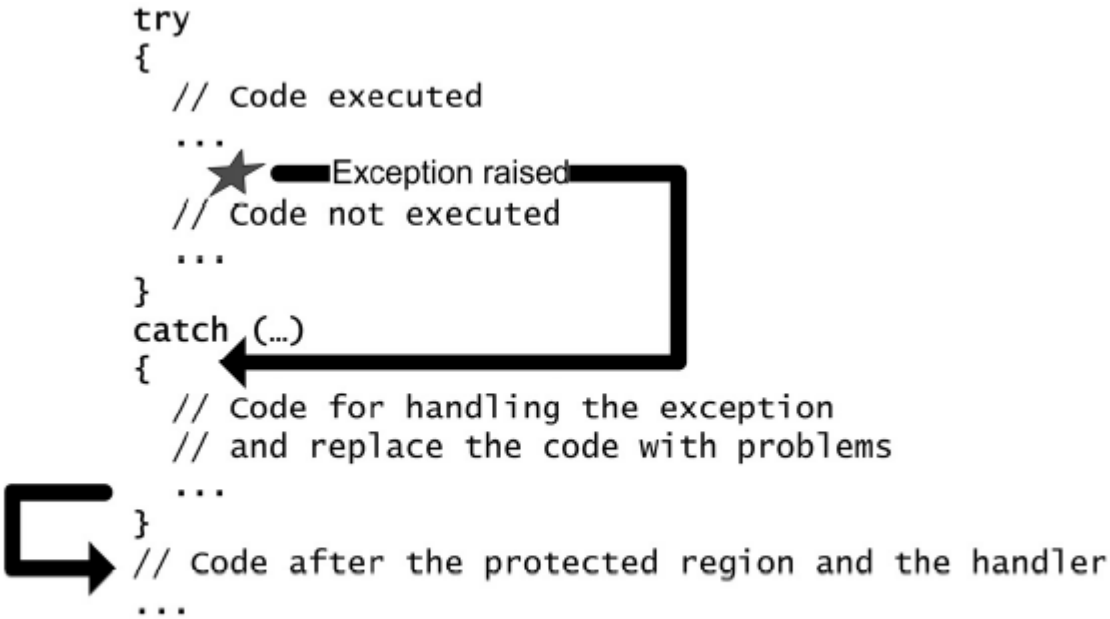


Figure 2.2 – Java code exemplifying the termination model

Ámbito de los manejadores en Java

```
try {
  _socket.send(_dp);
} catch(ConnectException ce) {
  // oh well, can't connect, ignore it...
} catch(BindException be) {
  // oh well, if we can't bind our socket, ignore it..
} catch(NoRouteToHostException nrthe) {
   // oh well, if we can't find that host, ignore it ...
} catch(IOException ioe) {
  if(isIgnoreable(ioe, ioe.getMessage()))
    return;
  String errString = "ip/port: " +
     _dp.getAddress() + ":" +
     _dp.getPort();
     _err.error(ioe, errString);
}
```

Figure 1 – Sample from LimeWire

Incremento en la complejidad

```
// The FileWriter must be declared outside of the try block
// and be pointing to something (null is a common choice)
FileWriter file = null;

try
{
    // Open file
    file = new FileWriter("data.txt");

    // Write some data into it
    for (int i=0; i<1024; i++)
        file.write("Here's the data: " + i);
}
catch (FileNotFoundException e)
{
    // Deal with filename problems
}
catch (SecurityException e)
{
    // Deal with problems like wrong permissions
}
catch (IOException e)
{
    // Deal with other (generic?) I/O problems
    // How do I do this???
}
finally
{
    try
    {
        file.close();
    }
    catch (IOException e)
    {
        // What should I do???
    }
}
```

Listing 1.2 – Writing to a file

Ámbito de los manejadores en Java

- La mejor solución al problema de la granularidad es permitir el paso de parámetros junto a las excepciones, Java lo hace automáticamente
- En Java la excepción es un objeto por tanto puede incluir tanta información como desee el programador

Ámbito de los manejadores en Java

- Se asemeja a una declaración de función, el parámetro es el tipo de excepción que atrapa
- Dentro del bloque el nombre del objeto es como una variable local
- Un manejador con parámetro de tipo T puede atrapar una excepción de tipo E si:
  - T y E son del mismo tipo
  - T es superclase de E (esto hace muy potente el mecanismo de manejo de excepciones de Java)

Ámbito de los manejadores en Java

```
        try {
                }
         catch (FileNotFoundException e) {
        System.err.println("FileNotFoundException:
"+ e.getMessage());//lanza excepción generada por
//el usuario
              throw new SampleException(e);}
        catch (IOException e) {//otra distinta de
//FileNotFoundException
System.err.println("Caught IOException: " +
e.getMessage());}
```

¿Cómo responder a los eventos anormales?

- Tareas típicas de un sistema de manejo de excepciones:
  - Registro (*logging*) y continuación: Los resultados anormales no necesariamente son malos desde el punto de vista del invocante. En el mejor caso puede registrarse las circunstancias y continuar con la operación normalmente
  - Registro y limitación de los daños: Registro para los programadores encargados del mantenimiento. Medidas para limitar daños: cierre de conexiones, reset de transacciones, etc. A menudo sólo es afectada una sesión y el usuario tendrá que darse de alta nuevamente; en otros casos, debe cerrarse (*shut down*) el sistema entero
  - Esperar (*timeout*) y/o repetir un número razonable de veces
  - Reconfiguración si existen componentes redundantes. Ej.: reemplazo una base de datos remota no accesible con una local (*backup*)

Exception Handling: A Field Study in Java and .NET

Table 3. Description of the Handler's actions categories.

Category	Description
Empty	The handler is empty, is has no code and does nothing more than cleaning the stack
Log	Some kind of error logging or user notification is carried out
Alternative/ Static Configuration	In the event of an error or in the execution of a finally block some kind of pre-determined (alternative) object state configuration is used
Throw	A new object is created and thrown or the existing exception is re-thrown
Continue	The protected block is inside a loop and the handler forces it to abandon the current iteration and start a new one
Return	The handler forces the method in execution to return or the application to exit. If the handler is inside a loop, a break action is also assumed to belong to this category
Rollback	The handler performs a rollback of the modifications performed inside the protected block or resets the state of all/some objects (e.g. recreating a database connection)
Close	The code ensures that an open connection or data stream is closed. Another action that belongs to this category is the release of a lock over some resource
Assert	The handler performs some kind of assert operation. This category is separated because it happens quite a lot. Note that in many cases, when the assertion is not successful, this results in a new exception being thrown possibly terminating the application
Delegates (only for .NET)	A new delegate is added
Others	Any kind of action that does not correspond to the previous ones

Resultados luego del manejo de excepciones

- Cada método tiene sólo dos posibles salidas:
  - Resultado normal, aún en el caso de errores de aplicación. El invocante ni se entera del uso de mecanismos de excepción, salvo por un tiempo de respuesta más largo
  - Avería final y definitiva: el método falla, se tomaron todas las medidas para limitar daños y se realizaron todos los intentos de reparación, cualquier intento posterior es inútil. La única opción restante para el invocante es abortar

Bloque *finally* en Java

- Java soporta una cláusula *finally* como parte de un bloque *try-catch*
- El código asociado a esta cláusula tiene garantizado su ejecución sin importar lo que ocurra con el que está en el bloque try
- Se utiliza para tareas de "limpieza". Es una herramienta clave para prevenir pérdidas de recursos

Bloque finally en Java

```
1   try {
2       file.open();
3       file.write("something");
4   } catch (IOException ioe) {
5       GUI.alertUser("Could not write to the file.");
6   } finally {
7       file.close();
8   }
```

Propagación de excepciones

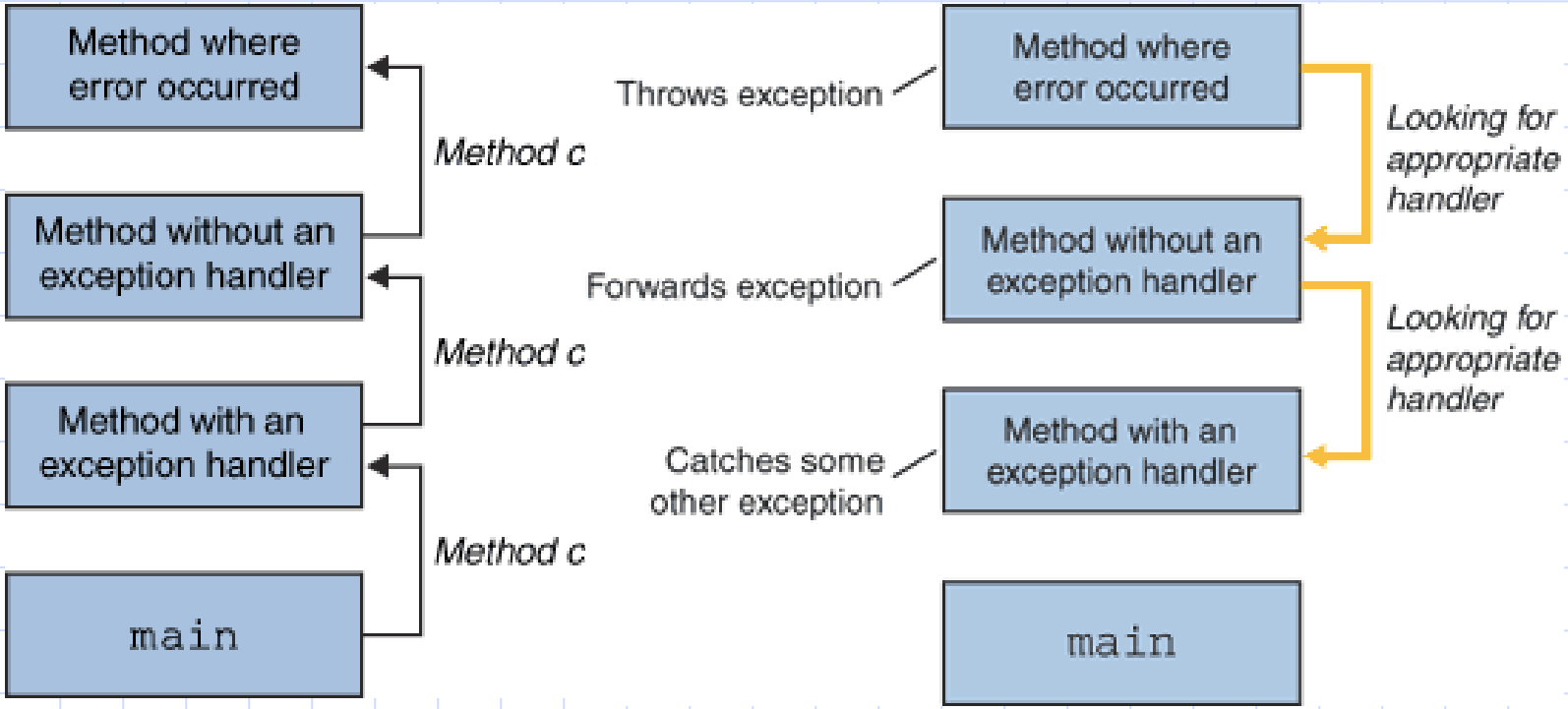
- Cuando no hay manejador en un bloque hay dos formas de proceder:
- Considerarlo un error de programación
  - Debe notificarse en tiempo de compilación.
  - Aunque una excepción generada en un procedimiento sólo pueda ser manejada apropiadamente en el contexto del invocante (excepción de interfaz)
  - Java permite que los procedimientos especifiquen qué excepciones pueden generar, es decir, cuáles no manejarán localmente, aunque, no requiere un manejador en el contexto del invocador

```
public static void main(String argv[]) throws IOException {

        BufferedReader input = new BufferedReader
                (new InputStreamReader(System.in));

String inputString = input.readLine();     // May throw IOException
                }
```

Propagación de excepciones

- Propagar la excepción (Java), es decir buscar manejadores en la cadena de invocaciones

Propagación de excepciones



Propagación de excepciones

- Si un bloque o procedimiento no tiene un manejador adecuado para una excepción generada, esta se propaga hacia el invocador
- Si ningún procedimiento en la cadena de llamadas tiene un manejador adecuado, la excepción es manejada por el *runtime*, abortando el programa
- Muchos lenguajes proveen un manejador de tipo ***catch all.***

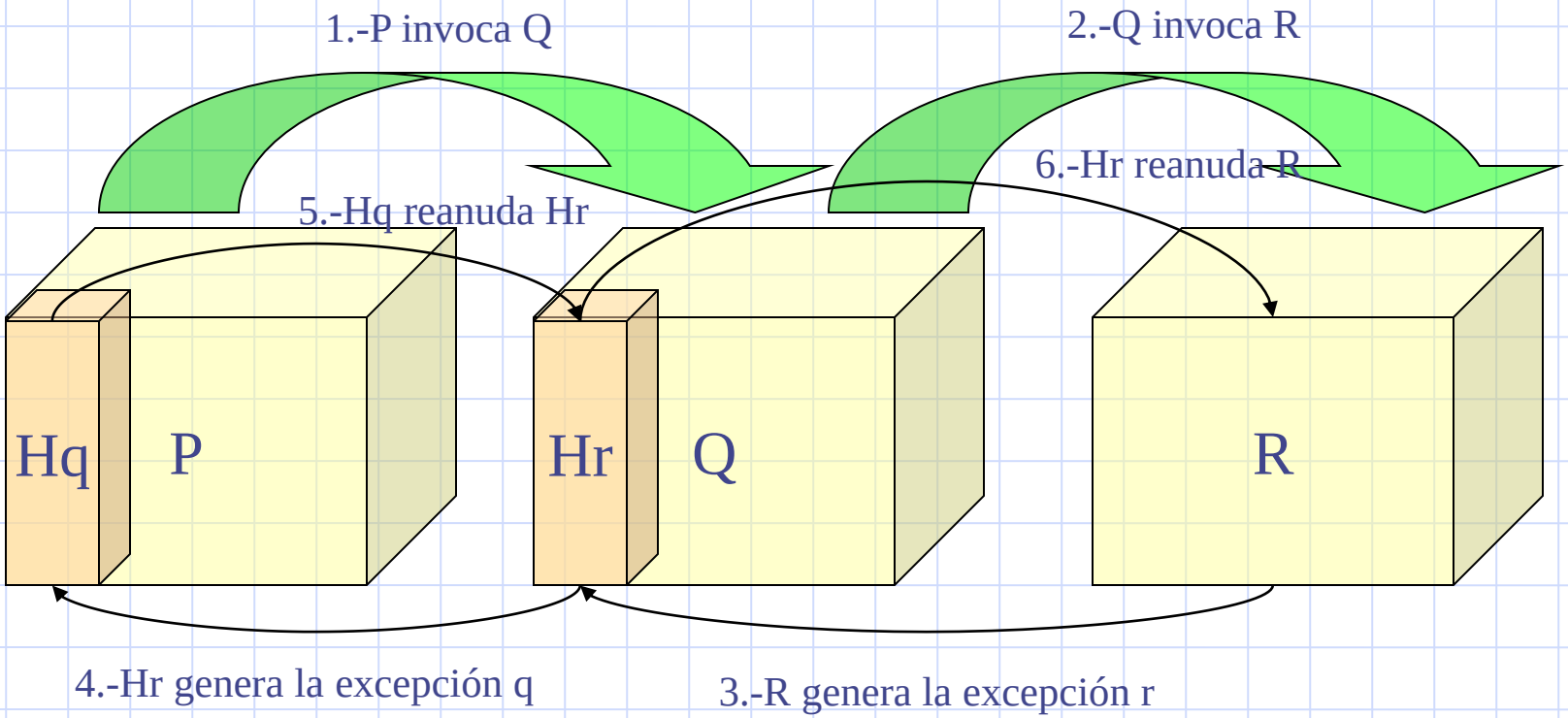
Catch all en Java

```
try {
   // statements which might raise the exception
   // IntegerConstraintError or ActuatorDead
}
catch(Exception E) {
   // handler will catch all exceptions of
   // type exception and any derived type;
   // but from within the handler only the
   // methods of Exception are accessible
}
```

Modelos de Tratamiento de Excepciones

- Siempre debemos determinar que conducta tomará el programa ante la presencia de una excepción
- Si el manejador resuelve el problema y el invocador (del gestor) puede continuar su ejecución, puede aplicarse el modelo de reanudación (o de notificación)
- Si no se devuelve el control al invocador el modelo se denomina de terminación o escape
- En un modelo híbrido el manejador puede decidir qué hacer

Modelo de Reanudación



Modelo de reanudación

- Ventaja: Cuando la excepción se generar asíncronamente, se puede reparar el daño y continuar (tiene poco que ver con el proceso en ejecución actualmente)
- Inconvenientes:
  - Es difícil reparar los errores detectados por el entorno de ejecución
  - Ejemplo: desborde aritmético en medio de una secuencia compleja de expresiones. Puede haber registros con evaluaciones parciales y el manejador puede sobreescribirlos
  - Es difícil de realizar: Puede ejecutarse nuevamente el bloque desde el principio. El manejador puede establecer un indicador local para señalar que se produjo un error (retry, pero las variables locales del bloque no deben reinicializarse en el reintento)

Modelo de reanudación

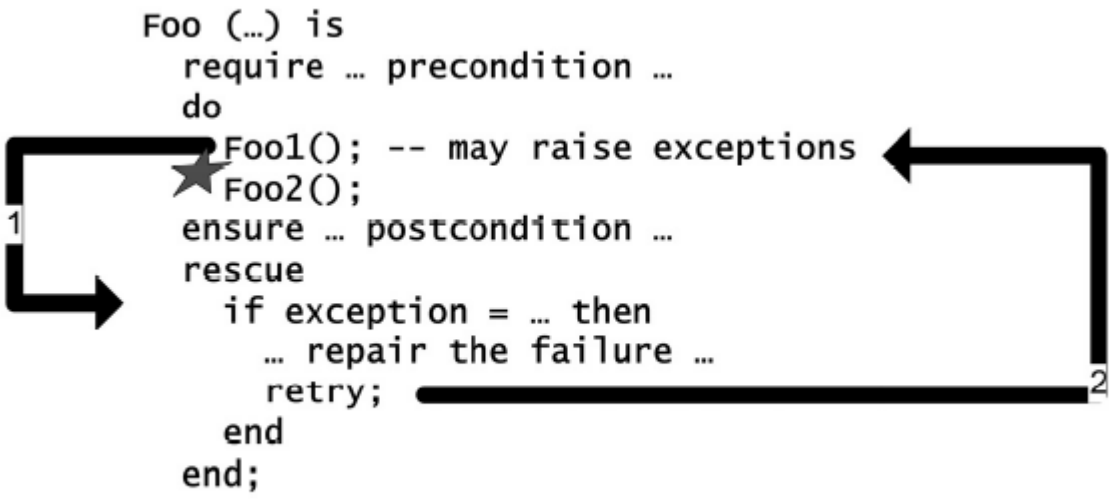
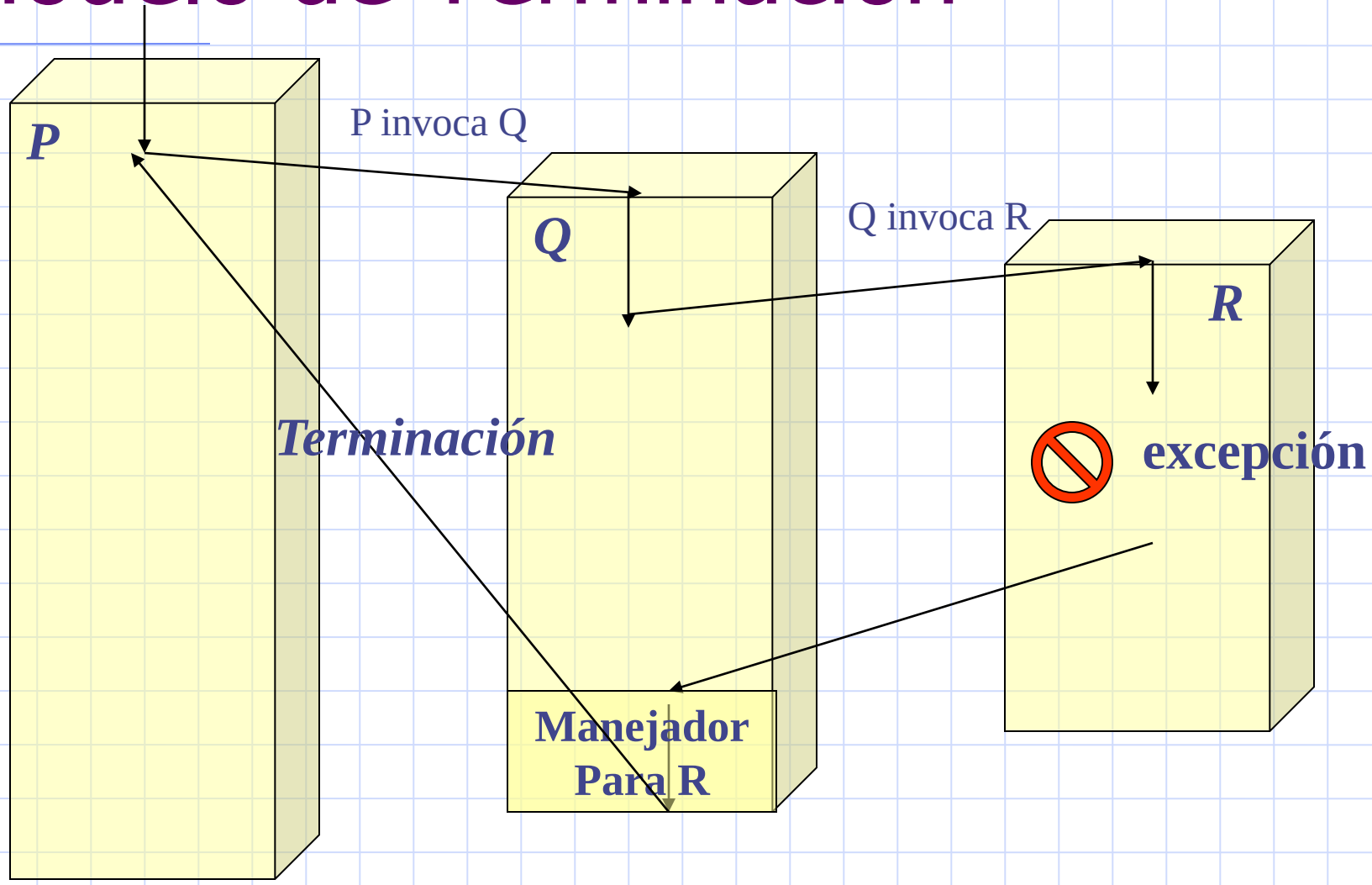


Figure 2.3 – Retry model exemplified with Eiffel notation

Modelo de Terminación



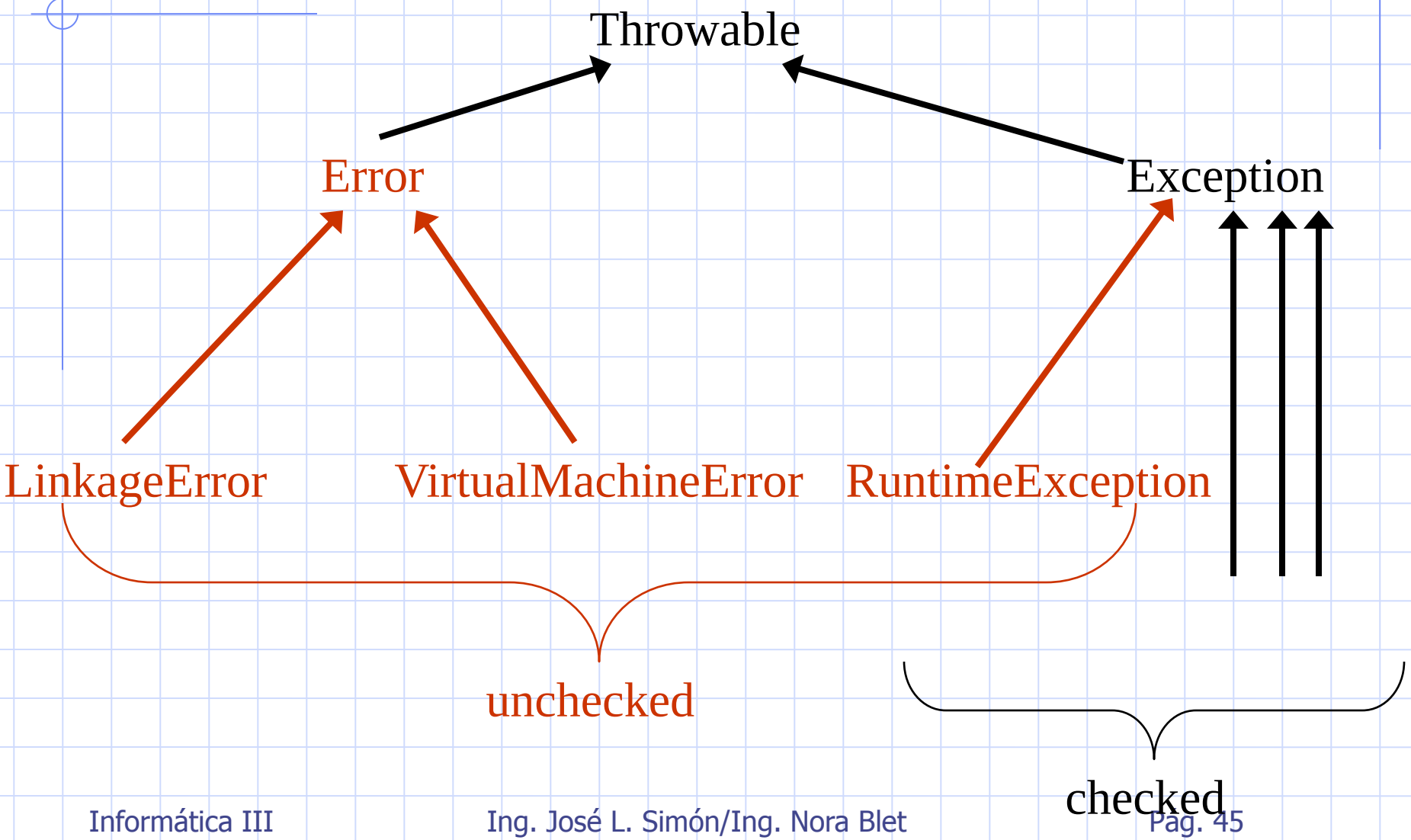
Modelo de terminación

- El control no retorna al punto donde apareció la excepción, sino que se finaliza el bloque o procedimiento que contiene el manejador y se pasa el control al bloque o procedimiento de llamada.
- Un procedimiento invocado puede terminar en condiciones: normal o de excepción
- Cuando el manejador está dentro de un bloque (Java) el control pasa a la primer sentencia que sigue al bloque que maneja la excepción

Excepciones en Java

- Son objetos de una clase (*Throwable*)
  - No hace falta declararlas explícitamente
  - El código de la aplicación o el entorno de ejecución pueden lanzar (*throw*) una excepción
  - El manejador apropiado la atrapa (*catch*). Debe hacer referencia a la clase o a una superclase
  - Utiliza el modelo de terminación para el manejo de excepciones

Las tres clases de excepciones en Java



Las tres clases de excepciones en Java

- Excepciones comprobadas (*checked exceptions*): condiciones excepcionales que pueden anticiparse y recuperarse de ellas. Ej.: incluir manejador para `java.io.FileNotFoundException` en código para manejar archivos
  - Son todas aquellas que no son del tipo `Error` o `RuntimeException` o subclases de ellas
  - Son manejadas en un bloque *try* o el método que las puede generar debe proveer una cláusula *throws* donde se las liste (**Catch or specify requirement**)

Las tres clases de excepciones en Java

- Error: condiciones excepcionales externas a la aplicación y usualmente ésta no puede anticiparlas ni recuperarse de ellas. Ej.: java.io.IOError, por problema de hard
  - Error y subclases
  - No están sujetos a los requerimientos de *try* o especificarlas en cláusulas *throws.* Igualmente pueden utilizarse

Las tres clases de excepciones en Java

- RunTimeException: Condiciones excepcionales internas a la aplicación y que usualmente la misma no puede anticipar o recuperarse de las mismas
  - Usualmente indican errores de programación (*bugs*), con lo cual tiene más sentido corregirlos que atraparlos. Ej.: NullPointerException en lugar de un nombre de archivo pasado a FileReader
  - Son de tipo RunTimeException o subclases
  - No están sujetos a los requerimientos de try o especificarlas en cláusulas throws. Igualmente pueden utilizarse
  - Las excepciones de este tipo y las de tipo Error son excepciones no comprobadas (unchecked exceptions)

Declaración

- Cada función debe declarar una lista de las excepciones verificadas que puede lanzar usando throws A, B, C. A, B y C son subclases de Exception
- La función puede lanzar (throw) cualquiera de las listadas y cualquier excepción no verificada
- Si la función intenta lanzar una excepción que no esté en la lista, se produce un error de compilación

Atrapar o especificar

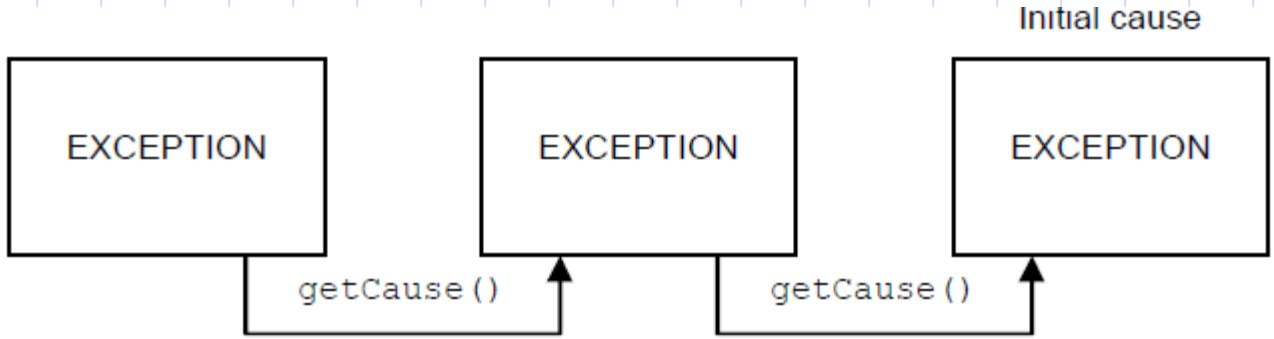
```
/* Reads the contents of a textfile. If the file does not exist,
 * creates an empty file and returns an empty string.
 */
public String readFile(String filename) throws IOException {
    String fileContents = "";
    FileReader filereader = null;
    try {
        filereader = new FileReader(filename);
    } catch (FileNotFoundException notfound) {
        new File(filename).createNewFile();
    }
    if (filereader != null) {
        BufferedReader reader = new BufferedReader(filereader);
        while (reader.ready()) {
            fileContents += reader.readLine();
        }
    }
    return fileContents;
}
```

Atrapar o especificar

- public FileReader(String fileName) throws FileNotFoundException
- public String readLine() throws IOException
- Cualquier función que invoque readFile debe atrapar la IOException o declarar (throws) que puede lanzarla

Excepciones encadenadas

- Una aplicación responde a una excepción lanzando otra, es decir la primer excepción causa la otra



Excepciones encadenadas

- try {
  }catch (IOException e) {
  throw new SampleException("Other IOException", e); }
- El stack trace provee información de la historia de la ejecución del thread actual y lista los nombres de las clases y métodos que fueron llamados en el punto donde ocurrió una excepción. Herramienta útil para hacer un debug, también permite construir un archivo de log

Referencias

- Burns, A. Wellings, A. "Sistemas de Tiempo Real y Lenguajes de Programación", Addison-Wesley (2003) Capítulo 6
- Transparencias de Juan Antonio de la Puente http://polaris.dit.upm.es/~jpuente/
- Siedersleben, J., Errors and Exceptions—Rights and Obligations, Lecture Notes in Computer Science, vol. 4119, pp. 275–287. Springer Berlin / Heidelberg, 2006
- Guerra, P. A. de C., Filho, F. C., Pagano, V. A. y Rubira, C. M. F., Structuring exception handling for dependable component-based software systems. EUROMICRO, pp. 575–582, 2004.
- B. Cabral, P. Marques, "Exception Handling: A field study in Java and .NET", in Proc. of the European Conference in Object-Oriented Programming 2007 (ECOOP'07), Berlin, Germany, July 2007.