Prácticas de Programación

Tema 4. Tratamiento de errores

4.1 Tratamiento de errores por paso de parámetros

En los lenguajes de programación más antiguos (C, Fortran, ...)

- Cada método o función retorna un valor
  - generalmente un código numérico o un booleano indicando si había habido error o no, y cuál

Dos inconvenientes:

- el código de chequeo de error aparece por todas partes, mezclado con el código normal
- en muchos casos el chequeo de error se omite por “pereza” o desconocimiento
  - lleva a situaciones de error que pasan inadvertidas

Ejemplo

El método `añadeAlumno` de una supuesta clase `Curso` sería:

```
/** añade un alumno. Retorna false si no se ha
 *  podido añadir */
public boolean añadeAlumno(Alumno a) {...}
```

Un fragmento de código que añade dos alumnos sería:

```
if (!añadeAlumno(a1)) {
  error.escribe("Error añadiendo alumno");
  return;
}
...
if (!añadeAlumno(a2)) {
  error.escribe("Error añadiendo alumno");
  return;
}
...
```

- el tratamiento de los errores hace difícil entender el código

4.2 Excepciones

Son un mecanismo especial para *gestionar errores*

- Permiten separar claramente 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

Las excepciones se pueden elevar (o lanzar):

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

Presentes en lenguajes “modernos” (Java, Ada, C++,...)

Ejemplo de elevación automática: División por cero

```
import fundamentos.*;
public class DivisionPorCero {
  public static void main(String[] args) {
    int i, j;
    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("i/j="+i/j);
    System.out.println("Fin");
  } // fin main
} // fin DivisionPorCero
```

cuando `j` vale 0 se eleva la excepción `ArithmeticException`

cuando se eleva la excepción esta línea no se ejecuta

Propagación de excepciones

1. Una línea de código genera o lanza (`throw`) una excepción
2. El bloque que contiene esa línea de código se aborta en ese punto
3. Si el bloque tiene un manejador para esa excepción, el manejador se ejecuta
   - diremos que el bloque ha cogido (`catch`) la excepción
   - la "vida" de la excepción finaliza en este punto
4. Si no tiene manejador, la excepción se propaga al bloque superior
   - que, a su vez, podrá cazar o dejar pasar la excepción
5. Si la excepción alcanza el bloque principal (`main`) y éste tampoco coge la excepción, el programa finaliza con un mensaje de error

Ejemplo: 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");
}
```


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)
```

4.3 Bloques de tratamiento 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 e) {
  instrucciones de tratamiento;
}
```

Los "`catch`" se evalúan por orden:

- **una excepción se coge en el primer "`catch`" para esa excepción o para una de sus superclases**

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: cazada " + e);
  }
}
```


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 cazada, 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

En el ejemplo anterior, el manejador realiza *únicamente* el tratamiento de la excepción `ArithmeticException`

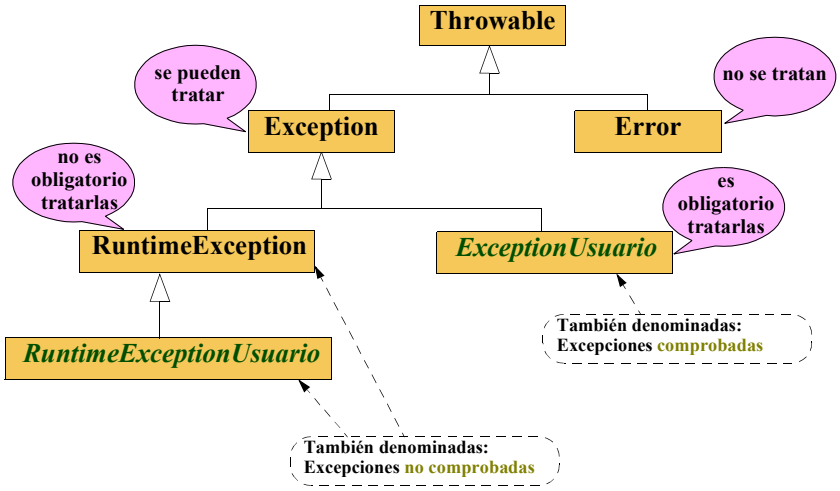
```
try {
   ...;
} catch (ArithmeticException e) {
   ...;
}
```

Es posible poner un *tratamiento común* para cualquier excepción

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

- **es cómodo pero *no es recomendable*, ya que puede ocurrir un tratamiento inadecuado para una excepción no prevista**

4.4 Jerarquía de las excepciones



Algunas excepciones `RuntimeException`

También se denominan excepciones *no comprobadas*

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

4.5 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

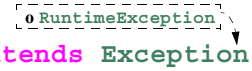
En algunas ocasiones un manejador puede volver a lanzar la misma excepción:

```
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**

4.6 Usar nuestras propias excepciones

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




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

Si queremos asociar mensajes con información adicional a una excepción nuestra, debemos escribir los constructores:

```
public static class MiExcepción extends Exception {
   // constructor sin argumentos
   public MiExcepción() {
     super();
   }
   // constructor con un string como argumento
   public MiExcepción(String infoAdicional) {
     super(infoAdicional);
   }
}
```

Excepciones "comprobadas" y cláusula `throws`

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

- **son *excepciones comprobadas***
- **(si extendieran `RuntimeException` serían "no comprobadas")**

Un método donde se lanza una excepción comprobada, deberá:

- **tratarla con un bloque `try-catch`**
- **o declarar que la lanza con una cláusula `throws`**

Sintaxis de la cláusula `throws`

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


Cláusula `throws` en métodos anidados

```
public void método3() throws MiExcepción {
  ...
  throw new MiExcepción();
}

public void método2() throws MiExcepción {
  ...
  método3();
}

public void método1() {
  ...
  try {
    método2();
  } catch (MiExcepción e) {
      ...
  }
}
```

lanza la excepción, por eso lo indica

no trata la excepción, por eso indica que puede lanzarla

trata la excepción, por eso no tiene cláusula `throws`

Ejemplo de uso de excepciones propias: `Curso`

Añadimos a la clase `Curso` la excepción `Completo`

- **elevada por el método `añadeAlumno` cuando no se pueden matricular más alumnos**
- **la clase *no proporciona ningún método que nos permita saber si el curso está completo* antes de añadir un alumno**

```
public class Curso {
  // lanzada por añadeAlumno cuando no se pueden
  // matricular más alumnos
  public class Completo extends Exception {}

  // lista de alumnos
  private final int MxAlumnos=100;
  private Alumno[] listaAlumnos=
                              new Alumno[MxAlumnos];
  private int numAlumnos=0;
```

```
   ...   // otros métodos

   /**
    * añade un alumno al curso.
    * @throws Completo cuando no se pueden
    * matricular más
    */
   public void añadeAlumno(Alumno a) throws Completo{
     if (numAlumnos == MxAlumnos) {
       throw new Completo();
     }
     listaAlumnos[numAlumnos] = a;
     numAlumnos++;
   }

   ...   // otros métodos
} // fin clase curso
```


Uso de `añadeAlumno` desde el programa principal (*la excepción es la única manera de detectar que el curso está completo*)

- **el que sea una excepción comprobada nos impide ignorar por error u olvido el posible error**

```
case AÑADIR:
  try {
    a = pideDatosAlumno();
    if (curso.buscaPorDNI(a.dni())!=null) {
      msj.escribe("El alumno ya existe");
    } else {
      curso.añadeAlumno(a); <- - - - - - - - - - Puede lanzar Curso.Completo
    }
  } catch (Curso.Completo e) {
     msj.escribe("No es posible añadir el " +
       "alumno, el curso está completo");
  }
  break;
```


Ejemplo de uso de excepciones propias: pila

Creamos excepciones para indicar los errores “pila llena” y “pila vacía”

```
public class PilaDeEnteros {
  // lanzada por apila cuando la pila está llena
  public class Llena extends RuntimeException {}
  // lanzada por desapila cuando la pila está vacía
  public class Vacía extends RuntimeException {}

  // pila de elementos
  private int[] pila;
  private int numElementos=0;

  /** Constructor. Recibe el número máximo de
   *  elementos que puede almacenar la pila */
  public PilaDeEnteros(int maxNumElementos) {
    pila = new int[maxNumElementos];
  }
```

```
/**
 * añade un elemento sobre la cima de la pila
 */
public void apila(int elemento) throws Llena {
  if (numElementos==pila.length)
    throw new Llena();
  pila[numElementos]=elemento;
  numElementos++;
}

/**
 * elimina el elemento en la cima de la pila
 */
public int desapila() throws Vacía {
  if (numElementos==0)
    throw new Vacía();
  numElementos--;
  return pila[numElementos];
}
```

Al tratarse de excepciones "No comprobadas" no es obligatorio poner el "throws", pero le ponemos por claridad


```
/**
 * retorna verdadero si la pila está vacía
 */
public boolean vacía() {
  return numElementos==0;
}

/**
 * retorna verdadero si la pila está llena
 */
public boolean llena() {
  return numElementos==pila.length;
}

} // fin clase PilaDeEnteros
```


Código para desapilar el elemento en la cima de la pila:

- **no necesitamos utilizar excepciones puesto que existe el método `pila.vacía()`**
  - **y puesto que `PilaDeEnteros.PilaVacía` es una excepción *no comprobada*, no es necesario el bloque `try-catch`**
- **aún así, la excepción serviría para detectar un uso incorrecto**

```
case DESAPILAR:
  if (pila.vacía()) {
    // caso especial: pila vacía
    System.out.println("Pila vacía");
    break;
  }
  // caso normal: pila con elementos
  int num = pila.desapila();
  System.out.println("<- Desapilado:" + num);
  break;
```

podría lanzar `PilaDeEnteros.Vacía` (excepción NO comprobada)

Ejemplo de uso de excepciones propias: pila (cont.)

Si `PilaDeEnteros.PilaVacía` fuera una excepción *comprobada*

- **nos obligaría a utilizar el bloque `try-catch` (aunque sólo serviría para complicar más el código)**

```
case DESAPILAR:
  try {
    if (pila.vacía()) {
      // caso especial: pila vacía
      System.out.println("Pila vacía");
      break;
    }
    // caso normal: pila con elementos
    int num = pila.desapila();
    System.out.println("<- Desapilado:" + num);
  } catch (PilaDeEnteros.Vacía e) {
    System.out.println("¡¡Nunca ocurre!!");
  }
  break;
```

podría lanzar `PilaDeEnteros.Vacía` (excepción comprobada)

4.7 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) {
   manejador1;
} catch (ClaseExcepción2 e) {
   manejador2;
} finally {
   siempre;
}
```

- **la cláusula `finally` es opcional**
- **todo `try` debe tener al menos un `catch` o un `finally`**

Secuencia de ejecución con cláusula `finally`

- **Si no hay error la secuencia es:**
  1. **operaciones**
  2. **siempre**
- **Si hay errores manejados la secuencia es:**
  1. **operaciones (incompletas)**
  2. **manejador**
  3. **siempre**
- **Si hay un error sin manejador:**
  1. **operaciones (incompletas)**
  2. **siempre**
  3. **se eleva la excepción en el siguiente bloque**

4.8 Utilización de excepciones

Las excepciones están pensadas para

- *tratar situaciones de error inesperadas o poco habituales*
- **para situaciones habituales es mejor retornar valores especiales (`null`, `false`, -1, ...)**

```
/**
 * Busca una palabra en el diccionario
 * @param palabra palabra de la que se busca el
 * significado
 * @return significado de la palabra o null en el
 * caso de que no esté en el diccionario
 */
 public String significado(String palabra) {...}
```

(En ocasiones determinar a priori si se trata de una situación excepcional o no puede ser difícil)

No deben **usarse las excepciones en casos que no sean de error**

- **por eficiencia**
- **y porque hacen más difícil entender el programa**

No debe **lanzarse una excepción y tratarla en el mismo método**

- **en lugar de lanzar la excepción, realizar el tratamiento directamente**
- **excepto si ya hay un manejador escrito que sea adecuado**

No deben **usarse excepciones para realizar el control de flujo**

- **por ejemplo para salirse de un lazo o una operación**
- **(ver ejemplos en las dos siguientes transparencias)**

Forma *correcta* de desapilar todos los elementos de la pila

- **la excepción no se utiliza para controlar el flujo de control**

```
case DESAPILAR_TODOS:

  if (pila.vacía()) {
    // caso especial: pila vacía
    System.out.println("Pila vacía");
    break;
  }

  // caso normal: pila con elementos
  do {
    int num=pila.desapila();
    System.out.println("<- Desapilado:"+num);
  } while (!pila.vacía());

  break;
```

podría lanzar `PilaDeEnteros.Vacía` (excepción NO comprobada)

Forma *incorrecta* de desapilar todos los elementos de la pila

- **la excepción se utiliza para controlar el flujo de control**

```
case DESAPILAR_TODOS_MAL:
  if (pila.vacía()) {
    System.out.println("Pila vacía");
    break;
  }
  do {
    try {
      int num=pila.desapila();
      System.out.println("<- Desapilado:"+num);
    } catch (PilaDeEnteros.Vacía e) {
      break;
    }
  } while (true);
```


Excepciones comprobadas vs. no comprobadas

Utilizaremos *excepciones comprobadas* cuando

- ***la excepción constituye la única manera de detectar el error***
- **y no queremos que pase inadvertido**
- **(ver la clase `Curso` en la página 21)**

Utilizaremos *excepciones no comprobadas* cuando

- ***la clase proporciona métodos para poder anticipar el error***
- **por lo tanto, si la clase se usa correctamente, la excepción no tendría porqué generarse nunca**
  - **los bloques `try-catch` serían superfluos**
- **la excepción sirve para detectar (y corregir) usos incorrectos de la clase**
- **(ver la clase Pila en la página 24)**

También utilizaremos *excepciones no comprobadas* para

- ***errores internos* ante los que el usuario de la clase poco puede hacer:**
  - **fallos en precondiciones de métodos privados**
  - **fallos en postcondiciones en métodos públicos o privados**

Reutilización de excepciones predefinidas

En ocasiones puede ser interesante reutilizar excepciones predefinidas

- **Ejemplo: reutilizamos la excepción `ArithmeticException` para indicar la división por el complejo (0,0)**

```
public class Complejo {
  // partes real e imaginaria del complejo
  private double r, i;

  /**
   * Constructor
   */
  public Complejo(double r, double i) {
    this.r = r;
    this.i = i;
   }
```



```
  /** retorna el número complejo resultado de la
   *  división: dividendo/divisor */
  public static Complejo división(Complejo dvdo,
                                  Complejo dvsor) {
    if (dvsor.r==0.0 && dvsor.i==0.0)
      throw new
         ArithmeticException("/ por Complejo(0,0)");

    Complejo cociente = new Complejo(0,0);
    cociente.r=(dvdo.r*dvsor.r + dvdo.i*dvsor.i)/
            (dvsor.r*dvsor.r + dvsor.i*dvsor.i);
    cociente.i=(-dvdo.r*dvsor.i + dvdo.i*dvsor.r)/
            (dvsor.r*dvsor.r + dvsor.i*dvsor.i);
    return cociente;
  }

  ...  // otros métodos
} // fin clase Complejo
```


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 la aplicación**
- ***recuperable*: se reintenta la operación**

- **Esquema de tratamiento de un error leve**

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

- **Esquema de tratamiento de un error grave**

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

- **Esquema de tratamiento de error recuperable**

```
while (true) {
   try {
      instrucciones a reintentar
      break o return
   } catch (ClaseExcepción e) {
      manejador
   }
}
```


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 lazo
  } catch (NumberFormatException e) {
    // no muestra mensaje de error porque ya
    // lo hace leeDouble
  }
} while (!notasCorrectas);
```


Ejemplo: lee dos notas en el rango [0.0, 10.0]

```
while (true) {
  lec.esperaYCierra("Introduce notas");
  try {
    nota1=lec.leeDouble("Nota parcial 1");
    nota2=lec.leeDouble("Nota parcial 2");
    if (nota1>=0.0 && nota1<=10.0 &&
        nota2>=0.0 && nota2<=10.0) {
      notasCorrectas = true; // sale del lazo
    } else {
      Mensaje error = new Mensaje();
      error.escribe("Rango incorrecto");
    }
  } catch (NumberFormatException e) {
    // no muestra mensaje de error porque ya
    // lo hace leeDouble
  }
} while (!notasCorrectas);
```

4.9 Bibliografía

- Ken Arnold, James Gosling, David Holmes, “El lenguaje de programación Java”, 3ª edición. Addison-Wesley, 2000.
- The Java Tutorials. “Lesson: Exceptions”. `http://java.sun.com/docs/books/tutorial/essential/exceptions/index.html`
- Francisco Gutiérrez, Francisco Durán, Ernesto Pimentel. “Programación Orientada a Objetos con Java”. Paraninfo, 2007.
- Eitel, Harvey M. y Deitel, Paul J., “Cómo programar en Java”, quinta edición. Pearson Educación, México, 2004.
- King, Kim N. “Java programming: from the beginning”. W. W. Norton & Company, cop. 2000