Manejo de errores

Principales conceptos a ser abordados

- Programación defensiva.
  - Anticipándose a que las cosas salgan mal
- Lanzamiento y manejo de excepciones.
- Informe de errores.
- Procesamiento simple de archivos.

Algunas causas de situaciones de error

- Implementación incorrecta.
  - No se ajusta a la especificación.
- Solicitar a un objeto una tarea que no puede realizar.
  - Por ej., índice no válido.
- Estado inconsistente o inapropiado de un objeto.
  - Por ej., aparece al reutilizar la extensión de una clase en otro proyecto.

Los errores no siempre los comete el programador

- Los errores a menudo llegan desde el ambiente:
  - Un URL ingresado incorrectamente.
  - Una interrupción de la red de datos.
- El procesamiento de archivos es generador de errores:
  - Archivos faltantes.
  - Falta de permisos apropiados.

Explorando errores

- Explorar las situaciones de error por medio de los proyectos:

  Libreta-de-direcciones.

- Dos aspectos:
  - Informe de errores.
  - Manejo de errores.

Programación defensiva

- Interacción cliente-servidor.
  - ¿Debería un servidor asumir que los clientes se comportan correctamente?
  - ¿O debería asumir que los clientes son potencialmente hostiles?
- Se requieren significativas diferencias en la implementación.

Puntos a ser considerados

- ¿Cuánta comprobación debe hacer el servidor en los llamados a métodos?
- ¿Cómo informar los errores?
- ¿Cómo puede un cliente anticipar un fallo?
- ¿Cómo debería un cliente tratar un fallo?

Un ejemplo

- Cree un objeto `LibretaDeDirecciones`.
- Trate de suprimir una entrada.
- Un error de ejecución resulta en...:
  - ¿De quién es este fallo?
- La anticipación y la prevención son preferibles que soportar reclamos.

Valores de los argumentos

- Los argumentos representan una gran ***vulnerabilidad*** para un objeto servidor.
  - Los argumentos de un constructor inicializan el estado.
  - Los argumentos de los métodos a menudo contribuyen al comportamiento.
- La comprobación de los argumentos es una de las medidas defensivas.

Comprobación de la clave

```
public void eliminarContacto(String clave)
{
    if(claveEnUso(clave)) {
        DatosDelContacto contacto = libreta.get(clave);
        libreta.remove(contacto.getNombre());
        libreta.remove(contacto.getTelefono());
        numeroDeEntradas--;
    }
}
```

Informe de errores del servidor

- ¿Cómo informar sobre argumentos ilegales?
  - ¿Al usuario?
    - ¿Hay un usuario humano?
    - ¿Puede resolver el problema?
  - ¿Al objeto cliente?
    - Retornar un valor diagnóstico.
    - *Arrojar una excepción*.

Retornar un diagnóstico

```
public boolean eliminarContacto(String clave)
{
    if(claveEnUso(clave)) {
        DatosDelContacto contacto = libreta.get(clave);
        libreta.remove(contacto.getNombre());
        libreta.remove(contacto.getTelefono());
        numeroDeEntradas--;
        return true;
    }
    else {
        return false;
    }
}
```

Respuestas del cliente

- Comprobar el valor retornado.
  - Intento de recuperarse del error.
  - Evitar la falla del programa.
- Ignorar el valor retornado.
  - No puede prevenirse.
  - Probabilidad de que se produzca la falla del programa.
- Las excepciones son preferibles.

Principios del lanzamiento de excepciones

- Es una característica especial del lenguaje.
- No se necesita retornar un valor ***especial***.
- El cliente no puede ignorar los errores.
- Se interrumpe el ***fujo-de-control normal***.
- Se recomiendan acciones específicas de recuperación.

Lanzamiento de una excepción

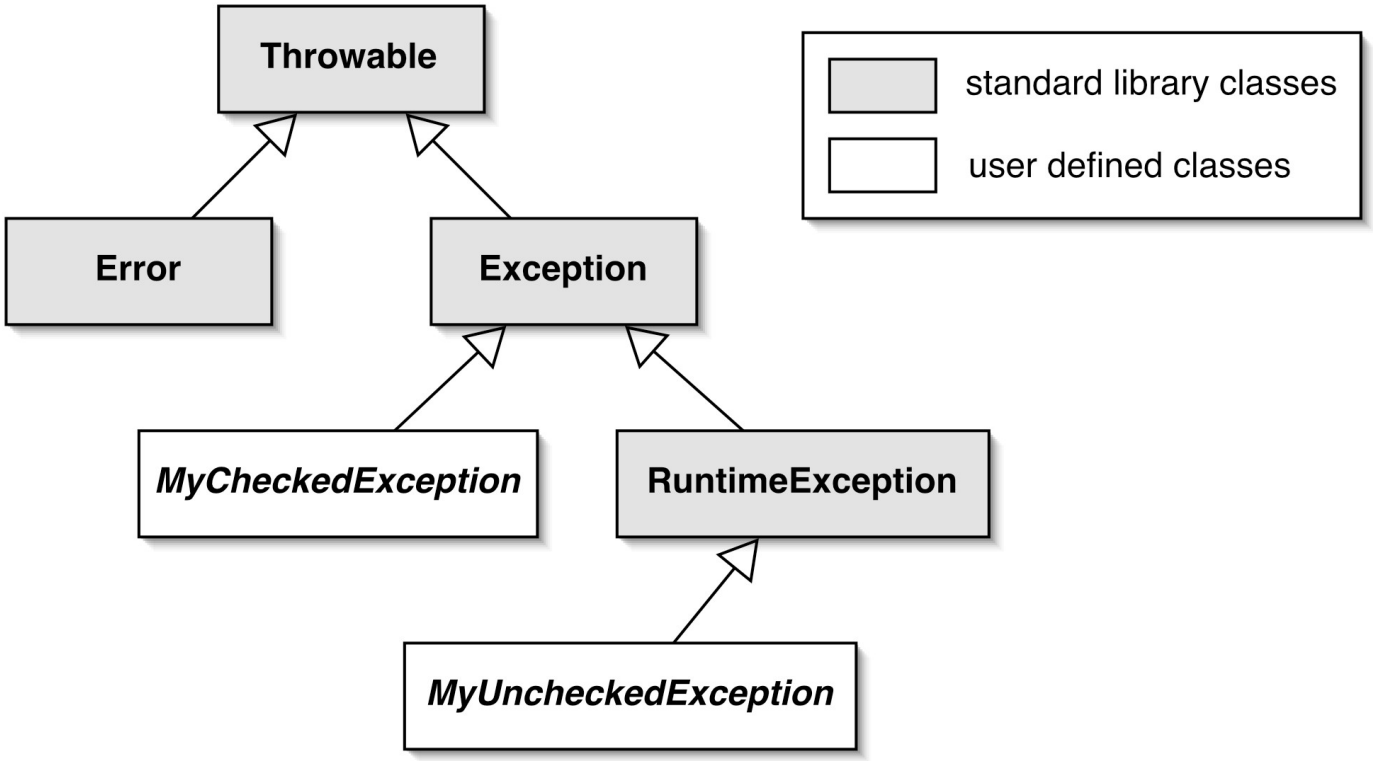
```
/**
 * Busca un nombre o un número de teléfono y devuelve
 * los datos de ese contacto.
 * @param clave El nombre o el número a buscar.
 * @return Los datos del contacto correspondiente a la
 * clave.
 */
    public DatosDelContacto getContacto(String clave)
    {
        if(clave == null) {
         throw new NullPointerException(
                                "clave null en getContacto");

        return libreta.get(clave);
    }
```

Lanzamiento de una excepción

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

La jerarquía de las clases excepción



Categorías de excepciones

- Excepciones comprobadas
  - Subclases de **`Exception`**
  - Usadas para fallos anticipados.
  - Cuando la recuperación pueda ser posible.
- Excepciones no comprobadas
  - Subclases de **`RuntimeException`**
  - Usadas para fallos no anticipados.
  - Cuando la recuperación no es posible.

El efecto de una excepción

- El método que genera el lanzamiento termina prematuramente.
- No se retorna un valor.
- El control no retorna al punto de llamada del cliente.
  - Luego, el cliente no puede llevar a cabo acciones de atención.
- Un cliente puede `catch` una excepción.

Excepciones no comprobadas

- Su uso **no** es **comprobado** por el compilador.
- Causan la terminación del programa si no se capturan.
  - Esta es la práctica normal.
- Un ejemplo típico es: **`IllegalArgumentException`**

Comprobación de argumentos

```
Public DatosDelContacto getContacto(String clave)
{
    if(clave == null) {
        throw new NullPointerException(
                        "clave null en getContacto");
    }
    if(clave.trim().length() == 0) {
        throw new IllegalArgumentException(
                        "clave null en getContacto");
    }
    return libreta.get(clave);
}
```

Impedir la creación de un objeto

```
public DatosDeContacto(String nombre, String telefono,
                       String direccion)
{
    if(nombre    == null) { name      = ""; }
    if(telefono  == null) { telefono  = ""; }
    if(direccion == null) { direccion = ""; }

    this.nombre    = nombre.trim();
    this.telefono  = telefono.trim();
    this.direccion = direccion.trim();

    if(this.nombre.length() == 0 && this.telefono.length() == 0) {
        throw new IllegalStateException(
                "El nombre y el teléfono no pueden estar vacíos.");
    }
}
```

Manejo de excepciones

- Se espera que las excepciones comprobadas se capturen.
- El compilador asegura que su uso se controle estrictamente.
  - Tanto en el servidor como en el cliente.
- Usadas adecuadamente, los fallos pueden ser recuperables.

La cláusula `throws`

- Los métodos que arrojan una excepción controlada deben incluir una cláusula **`throws`**:

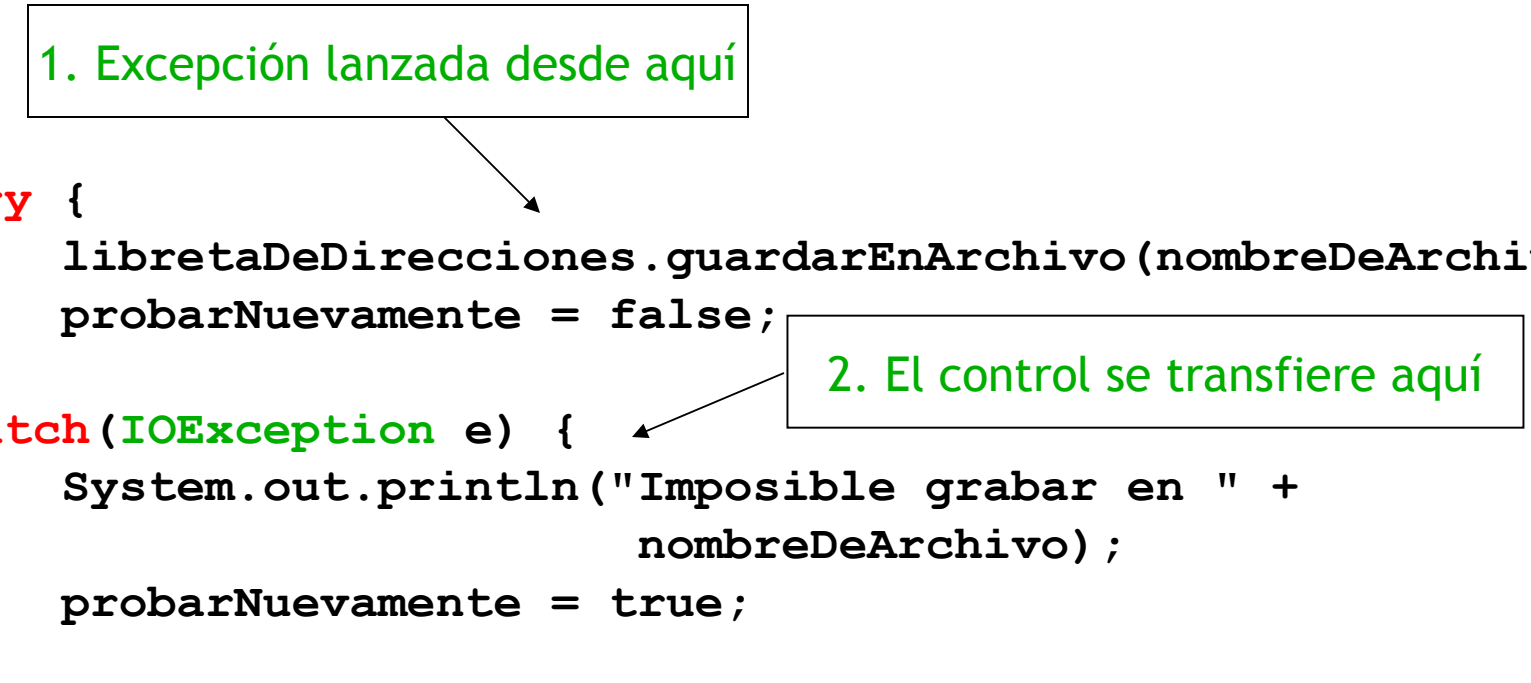
```
public void grabarEnArchivo(String archivoDestino)
    throws IOException
```

La sentencia `try`

- Los clientes que capturan una excepción deben proteger la llamada con una sentencia **`try`**:

```
try {
    Aqui se protegen una o más sentencias.
}
catch(Exception e) {
    Aquí se informa y se recupera de la exception.
}
```

La sentencia `try`




```
try {
    libretaDeDirecciones.guardarEnArchivo(nombreDeArchivo);
    probarNuevamente = false;
}
catch(IOException e) {
    System.out.println("Imposible grabar en " +
                       nombreDeArchivo);
    probarNuevamente = true;
}
```

Captura de excepciones múltiples

```
try {
    ...
    ref.procesar();
    ...
}
catch(EOFException e) {
    // Tomar las medidas adecuadas a una exception
    // fin de archivo (EOF).
    ...
}
catch(FileNotFoundException e) {
    // Tomar las medidas adecuadas a una exception
    // archivo no encontrado.
    ...
}
```

La cláusula `finally`

```
try {
    Aqí se protegen uno o más sentencias.
}
catch(Exception e) {
    Aquí se informa y se recupera de la excepción.
}
finally {
    Se realizan acciones comunes, se haya o no
    lanzado una excepción.
}
```

La cláusula `finally`

- Una cláusula **`finally`** se ejecuta aún si se ejecuta una sentencia **`return`** dentro de las cláusulas **`try`** o **`catch`**.
- Una excepción no capturada o propagada, aún sale por vía de la cláusula **`finally`**.

Definición de nuevas clases de excepciones

- *Extensión* de **`RuntimeException`** para una excepción no controlada o de **`Exception`** para una controlada.
- La definición de nuevos tipos de excepciones sirve para dar una mejor información diagnóstica.
  - Incluyendo información y/o recuperación.

```
public class NoCoincidenContactoException extends Exception
{
    private String clave;

    public NoCoincidenContactoException(String clave)
    {
        this.clave = clave;
    }

    public String getClave()
    {
        return clave;
    }

    public String toString()
    {
        return "No se econtraron datos que coincidan con: "
               + clave + ".";
    }
}
```

Aserciones

Se usan para comprobar la *consistencia interna*.

- Por ej.: el estado de un objeto después de una mutación.

- Normalmente, se usan durante el desarrollo y luego se quitan de la versión de producción.
  - Por ej.: vía una opción en tiempo de compilación.
- Java tiene una sentencia ***assert***.

La sentencia `assert` en Java

- Hay dos formas disponibles:
  - `assert` *expresion-booleana*
  - `assert` *expresion-booleana* : *cadena*
- La expresión-booleana verifica algo que debería ser verdad en este punto.
- Se lanza un `AssertionError` si la aserción es falsa.

La sentencia `assert` en Java

```
public void eliminarContacto(String clave)
    {
        if(clave == null){
            throw new IllegalArgumentException("...");
        }
        if(claveEnUso(clave)) {
            datosDelContacto contacto = libreta.get(clave);
            libreta.remove(contacto.getNombre());
            libreta.remove(contacto.getTelefono());
            numeroDeEntradas--;
        }
        assert !claveEnUso(clave);
        assert tamanioConsistente() :
                "El tamaño de la libreta es inconsistent" +
                " en eliminarContacto";
    }
```

Pautas para usar aserciones

- No son una alternativa al lanzamiento de excepciones.
- Usarlas para pruebas internas.
- Suprimirlas del código de producción.
- No incluir funcionalidad normal:

```
// Uso incorrecto:
assert libreta.remove(nombre) != null;
```

Recuperación de errores

- Los clientes deberían tomar nota de las notificaciones de errores.
  - Verificar los valores de retorno.
  - No *ignorar* las excepciones.
- Incluir código para intentar la recuperación.
  - A menudo requiere un ciclo de repetición.

Intento de recuperación

```
// Se intenta grabar la libreta de direcciones.
boolean exito = false;
int  intentos = 0;
do {
    try {
        libreta.grabarEnArchivo(nombreDeArchivo);
        successful = true;
    }
    catch(IOException e) {
        System.out.println("Imposible grabar en " +
                           nombreDeArchivo);
        intentos++;
        if(intentos < MAX_INTENTOS) {
            nombreDeArchivo = otro nombre de archivo;
        }
    }
} while(!exito && intentos < MAX_INTENTOSS);
if(!exito) {
    Informar el problema y rendirse;
}
```

Evitar los errores

- Los clientes pueden usar métodos de ***consulta al servidor*** para evitar los errores.
  - Clientes más robustos implican que los servidores pueden ser más confiables.
  - Pueden usarse las excepciones no comprobadas.
  - Simplifica la lógica del cliente.
- Puede incrementar el acoplamiento cliente-servidor.

Evitar una excepción

```
// Usar el método correcto para poner los
// detalles en la libreta de direcciones.
if(libreta.claveEnUso(contacto.getNombre() ||
    libreta.claveEnUso(contacto.getTelefono()) {
    libreta.modificarContacto(contacto);
}
else {
    libreta.agragarContacto(contacto);
}
```

El método **agregarContacto** ahora podría arrojar una excepción *no comprobada*.

Entrada/Salida de texto

- La entrada-salida es particularmente proclive al error.
  - Involucra la interacción con el ambiente externo.
- El paquete `java.io` soporta la entrada-salida.
- La `java.io.IOException` es una excepción comprobada.

Lectores, escritores y flujos

- Los lectores y los escritores tratan con la entrada y salida de texto.
  - Están basados en el tipo **`char`**.
- Los flujos tratan con datos binarios.
  - Están basados en el tipo **`byte`**.
- El proyecto *libreta-de-direcciones-io* ilustra la E/S de texto.

Salida de texto a un archivo

- Usar la clase **`FileWriter`**.
  - Abra un archivo.
  - Escriba en el archivo.
  - Cierre el archivo.
- Una falla en cualquier punto resulta en una **`IOException`**.

Salida de texto a archivo

```
try {
    FileWriter escritor = new FileWriter("nombre");
    while(hay más texto para escribir) {
        ...
        escritor.write(siguiente parte de texto);
        ...
    }
    escritor.close();
}
catch(IOException e) {
    Algo anduvo mal con al acceder al archivo.
}
```

Entrada de texto desde archivo

- Usar la clase **`FileReader`**.
- Aumentar con **`BufferedReader`** para entrada basada-en-líneas.
  - Abrir el archivo.
  - Leer desde el archivo.
  - Cerrar el archivo.
- Una falla en cualquier punto, resulta en una **`IOException`**.

Entrada de texto desde archivo

```
try {
    BufferedReader lector =
        new BufferedReader(
                new FileReader("nombrearchivo"));
    String linea = lector.readLine();
    while(linea != null) {
        Hacer algo con la linea
        linea = lector.readLine();
    }
    lector.close();
}
catch(FileNotFoundException e) {
    No se encontró el archivo especificado
}
catch(IOException e) {
    Algo salió mal al leer o cerrar el archivo
}
```

Entrada de texto desde archivo

- **`System.in`** mapea a la terminal.
  - **`java.io.InputStream`**
- A menudo envuelto en un
  - **`java.util.Scanner`**.
- **`Scanner`** soporta la separación en palabras (*parsing*) de la entrada de texto.
  - **`nextInt`**, **`nextLine`**, etc.
- **`Scanner`** con **`File`** es una alternativa a **`BufferedReader`** con **`FileReader`**.

Resumen

- Los errores en tiempo de ejecución ocurren por muchas razones.
  - Una llamada inapropiada de un cliente a un objeto servidor.
  - Un servidor incapaz de satisfacer un requerimiento.
  - Un error de programación en el cliente y/o el servidor.

Resumen

- Los errores en tiempo de ejecución a menudo llevan a una falla del programa.
- La programación defensiva anticipa los errores, tanto en el cliente como en el servidor.
- Las excepciones proveen un mecanismo de información y recuperación.