

Programación Orientada a Objetos VI - Manejo de errores y programación defensiva

Estructuras de Datos y Algoritmos /
Algoritmos y Estructuras de Datos II
Año 2025

Dr. Pablo Ponzio
Universidad Nacional de Río Cuarto
CONICET



Manejo de errores

- Cuando dos objetos interactúan existe la posibilidad de que algo salga mal, por una variedad de razones, por ejemplo:
 - El cliente invoca un método de un objeto en un estado que no satisface la precondición
 - Esto puede pasar porque el programador no comprendió correctamente como usar el objeto que provee el servicio (objeto servidor)
 - O el objeto cliente puede tener errores de programación que causan que le pase parámetros incorrectos al objeto servidor
 - El objeto servidor puede no ser capaz de cumplir con el requisito del cliente debido a ciertas circunstancias externas al objeto, como por ejemplo:
 - Errores de hardware (fallas en la conexión de red, fallas en el acceso a archivos en disco, etc.)
 - Entradas inválidas ingresadas por el usuario
 - Etc...
- Si queremos desarrollar aplicaciones confiables, debemos programarlas para que puedan recuperarse de estos problemas de manera elegante

Excepciones en violación de precondiciones

- Las excepciones pueden usarse para reportar violaciones de las precondiciones de los métodos
- Se lanza una excepción para reportarle al cliente que la invocación se realizó en un estado inválido
- Las excepciones se lanzan con la palabra reservada `throw`
- `throw` termina la ejecución del método y devuelve el control (y la excepción) al invocante
- Notar que las excepciones son objetos
- El tipo de la excepción, junto con el mensaje que se agrega a la misma, nos ayudan a entender la causa del problema
 - En el ejemplo, que el valor del parámetro es ilegal (`IllegalArgumentException`)
- Hay distintos tipos de excepciones, y el programador puede definir nuevos tipos

```
/**
 * @pre 'amount' > 0.
 * @post Receives an amount of money from a customer.
 */
public void insertMoney(int amount)
{
    if (amount <= 0)
        throw new IllegalArgumentException("amount must be positive");
    balance = balance + amount;
}
```

Excepciones en violación de precondiciones

- Notar que estas excepciones indican un error de programación en el cliente, que debe ser corregido
- Una forma de evitar estos problemas es chequear en el cliente que la precondición se satisface antes de invocar al método
- Usualmente, es preferible implementar estos chequeos en la lógica del cliente que capturando las excepciones (ver Figura)

```
/**
 * @pre 'amount' > 0.
 * @post Receives an amount of money from a customer.
 */
public void insertMoney(int amount)
{
    if (amount <= 0)
        throw new IllegalArgumentException("amount must be positive");

    balance = balance + amount;
}
```

```
// Cliente:
[...]  
do {  
    int amount = getUserInput();  
} while (amount <= 0);  
machine.insertMoney(amount);  
[...]
```

Excepciones en violación de precondiciones

- Lanzar este tipo de excepciones es crucial: identifican el lugar preciso de la falla, y lo reportan para que el problema pueda ser corregido en etapas tempranas del desarrollo
- Si no lo hacemos el error podría pasar desapercibido y las consecuencias pueden ser muy costosas
- Se pueden corromper datos importantes (bases de datos enteras), perder dinero, la aplicación puede crashear cuando está en producción, etc.

```
/**
 * @pre 'amount' > 0.
 * @post Receives an amount of money from a customer.
 */
public void insertMoney(int amount)
{
    if (amount <= 0)
        throw new IllegalArgumentException("amount must be positive");
    balance = balance + amount;
}
```

```
// Cliente:
[...]  
do {  
    int amount = getUserInput();  
} while (amount <= 0);  
machine.insertMoney(amount);  
[...]
```

Violación de precondiciones en la creación de objetos

- Si se produce una violación de una precondición en un constructor, podemos lanzar excepciones para prevenir la creación de objetos inválidos

```
/**
 * @pre 'cost' > 0.
 * @post Create a machine that issues tickets
 *       with a price of 'cost'.
 */
public TicketMachine(int cost)
{
    if (cost <= 0)
        throw new IllegalArgumentException("cost must be positive");

    price = cost;
    balance = 0;
    total = 0;
}
```

- Si invocamos al constructor con `cost <= 0` se lanza la excepción, se termina la ejecución del constructor y no se crea el objeto
- Notar como esto evita corromper el estado interno de nuestros programas, si permitimos crear el objeto con `cost == 0` estaríamos permitiendo crear máquinas que emiten tickets gratis

Excepciones en violación de precondiciones: Ejemplo

- Para la clase `ClockDisplay`:

```
/**
 * @pre 0 <= 'hour' < 24 and 0 <= 'minute' < 60.
 * @post Set the time of the display to the specified hour and
 * minute.
 */
public void setTime(int hour, int minute)
{
    if (hour < 0 || hour >= 24)
        throw new IllegalArgumentException("hour must be between 0 and 23");
    if (minute < 0 || minute >= 60)
        throw new IllegalArgumentException("minute must be between 0 and 60");

    hours.setValue(hour);
    minutes.setValue(minute);
    updateDisplay();
}
```

Excepciones en violación de precondiciones: Ejemplo

- Para la clase `MusicOrganizer`:

```
/**
 * @pre 0 <= 'index' < getNumberOfTracks()
 * @post Removes the track in position 'index' from the
 * collection.
 */
public void removeTrack(int index)
{
    if (index < 0 || index >= getNumberOfTracks())
        throw new IndexOutOfBoundsException("Invalid track index");
    tracks.remove(index);
}
```


No pasar `null` como parámetro

- `null` es un valor problemático, ya que en principio cualquier objeto de cualquier tipo puede ser `null`
- Por lo tanto, tendríamos que agregar código a cada método que tome objetos como parámetros para chequear explícitamente que cada objeto no sea `null`


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

- Y vamos a tener que agregar que los objetos sean distintos de `null` en todas las precondiciones que lo requieran
- Esto termina ensuciando todo el código, y es prohibitivo en proyectos grandes
 - Y suele ser poco efectivo para encontrar errores

No pasar `null` como parámetro

- Para evitar esto, vamos a tomar una como política general de programación nunca pasar `null` como parámetro
 - Tomada del libro Clean Code [1]
- Y en general, vamos a evitar usar `null` tanto como sea posible
- De esta manera, nos ahorramos todos estos chequeos que terminan ensuciando el código y haciendo más tedioso nuestro trabajo como programadores

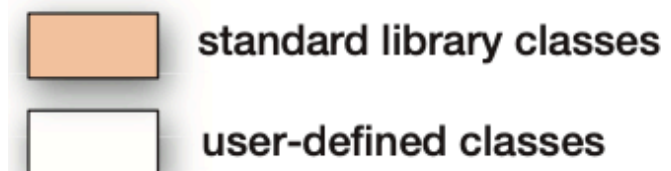
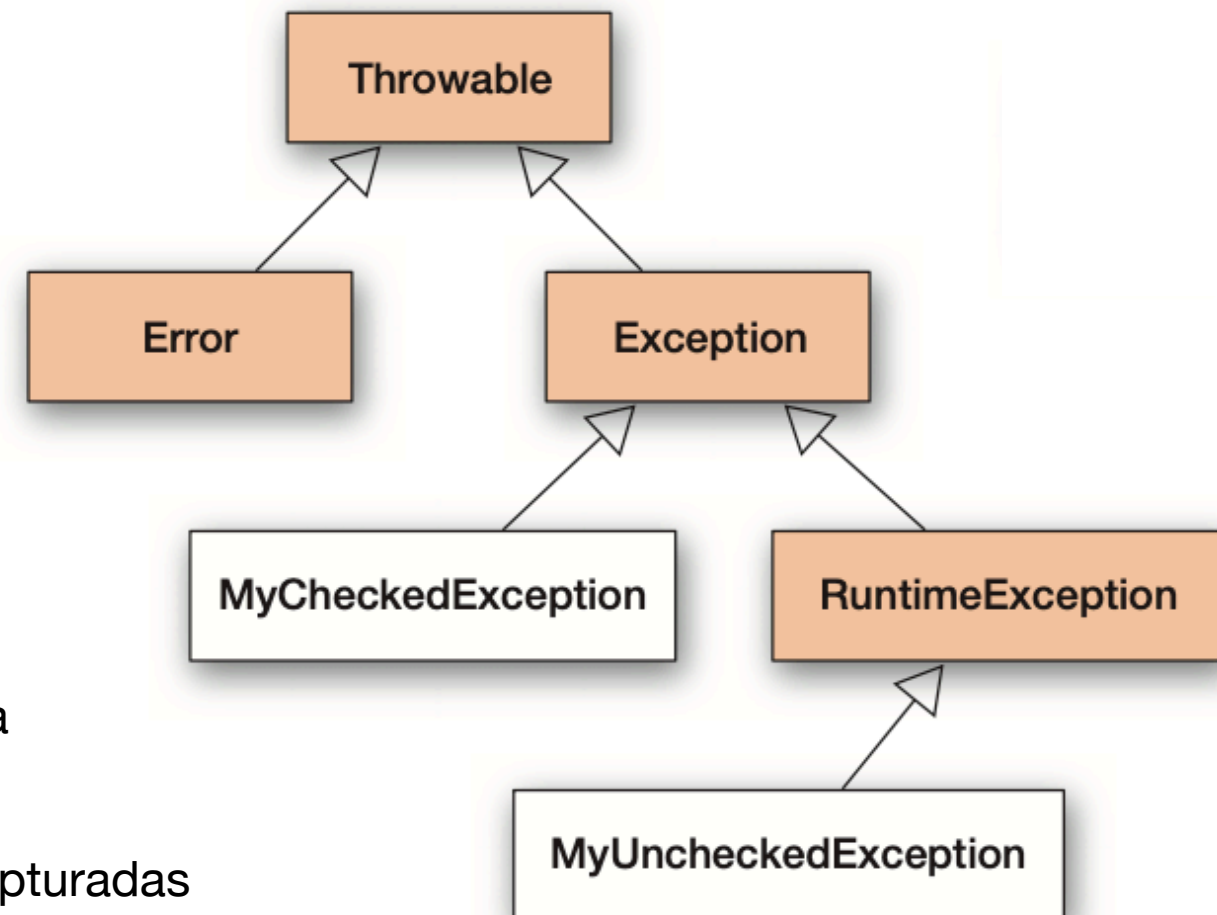
```
public ContactDetails getDetails(String key)
{
    if(key == null) {
        throw new IllegalArgumentException("null key in getDetails");
    }
    if(key.trim().length() == 0) {
        throw new IllegalArgumentException(
            "Empty key passed to getDetails");
    }
    return book.get(key);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



- Veremos que con un poco de esfuerzo de diseño y de programación no es difícil evitar usar `null` casi por completo

Excepciones verificadas y no verificadas

- La figura muestra la jerarquía de clases en la que se organizan las excepciones en Java
- `Exception` es la clase de la que heredan las excepciones que la mayoría de las aplicaciones usan
- Las excepciones que heredan de `Exception` se denominan excepciones verificadas (checked exceptions)
- Las excepciones que heredan de `RuntimeException` son excepciones no verificadas (unchecked exceptions)
- Las excepciones verificadas imponen dos requisitos:
 - Deben ser declaradas en el perfil del método que las lanza
 - El código del cliente del método debe capturarla
- El compilador verifica este tipo de excepciones, y si no son capturadas el programa no compila
- En cambio, las excepciones no verificadas no imponen ningún requisito sobre los clientes
- Por último, `Error` indica errores serios en la ejecución de Java sobre los cuales los programadores usualmente no tienen control (ej. errores de threads)
 - La mayoría de las aplicaciones no deberían intentar capturarlos



Excepciones verificadas:

Ejemplo

- Como ejemplo de excepciones verificadas, usaremos el proyecto `weblog-analyzer`
- Su propósito es crear archivos con fechas aleatorias para testear un analizador de archivos de logs
- Para esto, la clase `LogfileCreator` define el siguiente método:

```
/**
 * @pre 'numEntries' > 0 &&
 *      'filename' is a valid filename.
 * @post Create a file of random log with name 'filename' and
 *      'numEntries' entries. Throws an IOException if there are errors
 *      when trying to write to 'filename'.
 */
public void createFile(String filename, int numEntries) throws IOException
```

- La creación de archivos puede fallar por diversas razones: el nombre de archivo es inválido en el sistema operativo, no hay permisos para escribir en el archivo, falla en el disco rígido, etc...
- Para indicar que ocurrió uno de estos problemas, el método lanzará una `IOException`

Excepciones verificadas:

Ejemplo

```
2016 02 27 13 53
2016 04 14 21 55
2016 05 10 00 19
2016 06 09 18 49
2016 06 11 20 15
2016 09 01 23 01
2016 09 11 13 30
2016 10 14 16 19
2016 10 18 17 17
2016 11 11 02 02
```

- Como ejemplo de excepciones verificadas, usaremos el proyecto `weblog`
- Su propósito es crear archivos con fechas aleatorias para testear un analizador de archivos de logs
- Para esto, la clase `LogfileCreator` define el siguiente método:

```
/**
 * @pre 'numEntries' > 0 &&
 *      'filename' is a valid filename.
 * @post Create a file of random log with name 'filename' and
 *      'numEntries' entries. Throws an IOException if there are errors
 *      when trying to write to 'filename'.
 */
public void createFile(String filename, int numEntries) throws IOException
```

logfile.txt

- La creación de archivos puede fallar por diversas razones: el nombre de archivo es inválido en el sistema operativo, no hay permisos para escribir en el archivo, falla en el disco rígido, etc...
- Para indicar que ocurrió uno de estos problemas, el método lanzará una `IOException`

Excepciones verificadas:

Ejemplo

```
2016 02 27 13 53
2016 04 14 21 55
2016 05 10 00 19
2016 06 09 18 49
2016 06 11 20 15
2016 09 01 23 01
2016 09 11 13 30
2016 10 14 16 19
2016 10 18 17 17
2016 11 11 02 02
```

- Como ejemplo de excepciones verificadas, usaremos el proyecto `weblog`
- Su propósito es crear archivos con fechas aleatorias para testear un analizador de archivos de logs
- Para esto, la clase `LogfileCreator` define el siguiente método:

```
/**
 * @pre 'numEntries' > 0 &&
 *      'filename' is a valid filename.
 * @post Create a file of random log with name 'filename' and
 *      'numEntries' entries. Throws an IOException if there are errors
 *      when trying to write to 'filename'.
 */
public void createFile(String filename, int numEntries) throws IOException
```

logfile.txt

Excepción que el invocante está forzado a capturar

- La creación de archivos puede fallar por diversas razones: el nombre de archivo es inválido en el sistema operativo, no hay permisos para escribir en el archivo, falla en el disco rígido, etc...
- Para indicar que ocurrió uno de estos problemas, el método lanzará una `IOException`

Excepciones verificadas:

Ejemplo

```
/**
 * @pre 'numEntries' > 0 &&
 *      'filename' is a valid filename.
 * @post Create a file of random log with name 'filename' and
 *      'numEntries' entries. Throws an IOException if there are errors
 *      when trying to write to 'filename'.
 */
public void createFile(String filename, int numEntries) throws IOException
{
    if (numEntries <= 0)
        throw new IllegalArgumentException("numEntries should be greater than 0.");

    FileWriter writer = new FileWriter(filename);
    LogEntry[] entries = new LogEntry[numEntries];
    for(int i = 0; i < numEntries; i++) {
        entries[i] = createEntry();
    }
    Arrays.sort(entries);
    for(int i = 0; i < numEntries; i++) {
        writer.write(entries[i].toString());
        writer.write('\n');
    }

    writer.close();
}
```


Excepciones verificadas:

Ejemplo

```
/**
 * @pre 'numEntries' > 0 &&
 *      'filename' is a valid filename.
 * @post Create a file of random log with name 'filename' and
 *      'numEntries' entries. Throws an IOException if there are errors
 *      when trying to write to 'filename'.
 */
public void createFile(String filename, int numEntries) throws IOException
{
    if (numEntries <= 0)
        throw new IllegalArgumentException("numEntries should be greater than 0.");

    FileWriter writer = new FileWriter(filename);
    LogEntry[] entries = new LogEntry[numEntries];
    for(int i = 0; i < numEntries; i++) {
        entries[i] = createEntry();
    }
    Arrays.sort(entries);
    for(int i = 0; i < numEntries; i++) {
        writer.write(entries[i].toString());
        writer.write('\n');
    }

    writer.close();
}
```



Lanza una
IOException si no
puede crear el
archivo

Excepciones verificadas:

Ejemplo

```
/**
 * @pre 'numEntries' > 0 &&
 *      'filename' is a valid filename.
 * @post Create a file of random log with name 'filename' and
 *      'numEntries' entries. Throws an IOException if there are errors
 *      when trying to write to 'filename'.
 */
public void createFile(String filename, int numEntries) throws IOException
{
    if (numEntries <= 0)
        throw new IllegalArgumentException("numEntries should be greater than 0.");

    FileWriter writer = new FileWriter(filename);
    LogEntry[] entries = new LogEntry[numEntries];
    for (int i = 0; i < numEntries; i++) {
        entries[i] = new LogEntry();
    }
    Arrays.sort(entries);
    for (int i = 0; i < numEntries; i++) {
        writer.write(entries[i].toString());
        writer.write('\n');
    }

    writer.close();
}
```

Lanza una IOException si no puede crear el archivo

Lanza una IOException si no puede escribir en el archivo

Excepciones verificadas:

Ejemplo

```
/**
 * @pre 'numEntries' > 0 &&
 *      'filename' is a valid filename.
 * @post Create a file of random log with name 'filename' and
 *      'numEntries' entries. Throws an IOException if there are errors
 *      when trying to write to 'filename'.
 */
public void createFile(String filename, int numEntries) throws IOException
{
    if (numEntries <= 0)
        throw new IllegalArgumentException("numEntries should be greater than 0.");

    FileWriter writer = new FileWriter(filename);
    LogEntry[] entries = new LogEntry[numEntries];
    for (int i = 0; i < numEntries; i++) {
        entries[i] = new LogEntry();
    }
    sort(entries);
    for (int i = 0; i < numEntries; i++) {
        write(entries[i].toString());
        write('\n');
    }
    writer.close();
}
```

Lanza una
IOException si no
puede crear el
archivo

Lanza una
IOException si no
puede escribir en el
archivo

Lanza una
IOException si no
puede cerrar el
archivo

Excepciones verificadas:

Ejemplo

```
/**
 * @pre 'numEntries' > 0 &&
 *      'filename' is a valid filename.
 * @post Create a file of random log with name 'filename' and
 *      'numEntries' entries. Throws an IOException if there are errors
 *      when trying to write to 'filename'.
 */
public void createFile(String filename, int numEntries) throws IOException
{
    if (numEntries <= 0)
        throw new IllegalArgumentException("numEntries should be greater than 0.");

    FileWriter writer = new FileWriter(filename);
    LogEntry[] entries = new LogEntry[numEntries];
    for (int i = 0; i < numEntries; i++) {
        entries[i] = new LogEntry();
    }
    sort(entries);
    for (int i = 0; i < numEntries; i++) {
        writer.write(entries[i].toString());
        writer.write('\n');
    }
    writer.close();
}
```

Lanza una
IOException si no
puede crear el
archivo

Lanza una
IOException si no
puede escribir en el
archivo

Lanza una
IOException si no
puede cerrar el
archivo

Consultar la documentación de FileWriter:
<https://docs.oracle.com/javase/8/docs/api/java/io/FileWriter.html>

Capturando excepciones

- En Java, para tratar con código que puede lanzar excepciones usamos la sentencia `try`
 - Este código se llama manejador de excepciones

```
try {  
    Protect one or more statements here.  
}  
catch(Exception e) {  
    Report and recover from the exception here.  
}
```

- El bloque del `try` abarca las sentencias que pueden lanzar una excepción
- Si alguna de las sentencias del bloque `try` lanza una excepción, y la excepción matchea con el `catch`, se ejecuta el bloque del `catch`
 - El `catch` permite intentar recuperarnos de una excepción/reportar el error de manera precisa/salir del programa elegantemente
 - Si es posible recuperar el programa, la ejecución debe continuar con la instrucción que sigue luego del `catch`
- Si la excepción no matchea con el `catch`, se retorna la excepción al invocante

LogfileCreator: Cliente

- Notar que las instrucciones dentro del `catch` sólo se ejecutan si `createFile` lanza una excepción de tipo `IOException`
- En tal caso, le informamos al usuario del error
- Y el ciclo volverá a reintentar la creación de un nuevo archivo
- En caso de que `createFile`, no lance excepciones, asignamos `true` a `success`, las instrucciones dentro del `catch` no se ejecutan, y el ciclo termina

[illegible]

LogfileCreator: Cliente mejorado

- Por razones de seguridad, no es buena idea dejar que el input del usuario pase directamente a la creación de archivos sin ninguna verificación
- Un usuario malicioso podría ingresar datos en archivos no permitidos de nuestro disco
- Por esta razón, y para cumplir mejor con la precondición de `createFile`, vamos a agregar algunos chequeos adicionales sobre el input del usuario antes de intentar crear el archivo
- Estos chequeos se implementan en el método `isValidPath`

```
boolean success = false;
while (!success) {
    String filename = reader.readUserInput();
    // Better security checks to satisfy createFile's
    // precondition.
    if (isValidPath(filename))
    {
        try
        {
            creator.createFile(filename, LOG_ENTRIES);
            success = true;
        }
        // Do nothing, success is already false.
        catch (IOException exception) { }
    }
    if (!success) {
        System.out.println("Error writing to file." +
            " Try again.");
    }
}
```



```

/**
 * @pre The path is in a valid directory (TODO: not checked).
 * @post Returns true if and only if 'path' is a
 * valid path for the user's operating system.
 *
 * Calling examples on Windows:
 *     isValidPath("c:/test");           //returns true
 *     isValidPath("c:/te:t");           //returns false
 *     isValidPath("c:/te?t");           //returns false
 *     isValidPath("c/te*t");            //returns false
 *     isValidPath("good.txt");           //returns true
 *     isValidPath("not|good.txt");       //returns false
 *     isValidPath("not:good.txt");       //returns false
 */
public static boolean isValidPath(String path)
{
    // TODO: To avoid security problems, we should also verify
    // that the file belongs to the directory we want to use
    // to store log files
    try
    {
        Path javaPath = Paths.get(path);
    } catch (InvalidPathException | NullPointerException ex)
    {
        return false;
    }
    return true;
}

```

```

/**
 * @pre The path is in a valid directory (TODO: not checked).
 * @post Returns true if and only if 'path' is a
 * valid path for the user's operating system.
 *
 * Calling examples on Windows:
 *     isValidPath("c:/test");           //returns true
 *     isValidPath("c:/te:t");           //returns false
 *     isValidPath("c:/te?t");           //returns false
 *     isValidPath("c/te*t");            //returns false
 *     isValidPath("good.txt");          //returns true
 *     isValidPath("not|good.txt");      //returns false
 *     isValidPath("not:good.txt");      //returns false
 */

```

```

public static boolean isValidPath(String path)
{

```

```

    // TODO: To avoid security problems, we should also verify
    // that the file belongs to the directory we want to use
    // to store log files

```

```

    try
    {
        Path javaPath = Paths.get(pat
    } catch (InvalidPathException | NullPointerException ex)
    {
        return false;
    }
    return true;
}

```

Se puede capturar más de un tipo de excepción en un catch


```

/**
 * @pre The path is in a valid directory (TODO: not checked).
 * @post Returns true if and only if 'path' is a
 * valid path for the user's operating system
 *
 * Calling examples on Windows:
 *   isValidPath("c:/test");
 *   isValidPath("c:/te:t");           //returns false
 *   isValidPath("c:/te?t");           //returns false
 *   isValidPath("c/te*t");             //returns false
 *   isValidPath("good.txt");           //returns true
 *   isValidPath("not|good.txt");       //returns false
 *   isValidPath("not:good.txt");       //returns false
 */

```

Este método no es perfecto, ya que permite crear archivos en cualquier directorio en el que el usuario tenga permisos

```

public static boolean isValidPath(String path)
{

```

```

    // TODO: To avoid security problems, we should also verify
    // that the file belongs to the directory we want to use
    // to store log files

```

```

    try
    {
        Path javaPath = Paths.get(path);
    } catch (InvalidPathException | NullPointerException ex)
    {
        return false;
    }
    return true;
}

```

Se puede capturar más de un tipo de excepción en un catch

```

/**
 * @pre The path is in a valid directory (TODO: not checked).
 * @post Returns true if and only if 'path' is a
 * valid path for the user's operating system
 *
 * Calling examples on Windows:
 *   isValidPath("c:/test");
 *   isValidPath("c:/te:t");           //returns false
 *   isValidPath("c:/te?t");           //returns false
 *   isValidPath("c/te*t");             //returns false
 *   isValidPath("good.txt");           //returns true
 *   isValidPath("not|good.txt");       //returns false

```

Este método no es perfecto, ya que permite crear archivos en cualquier directorio en el que el usuario tenga permisos

Ejercicio: Mejorar `isValidPath` para verificar que sólo permita crear archivos dentro de un directorio `logs` en la carpeta raíz del proyecto

```

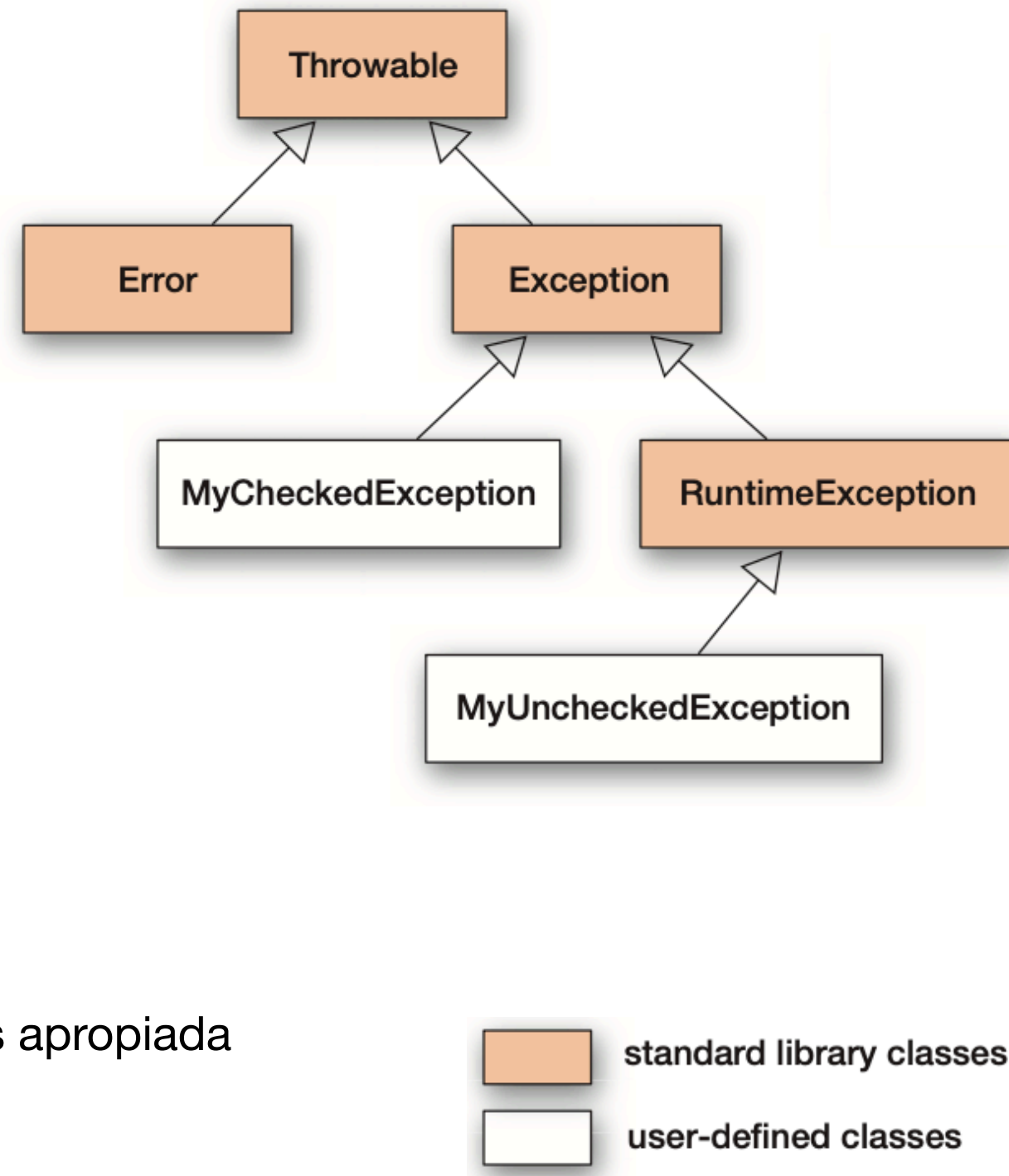
{
    // TODO: To avoid security problems, we should also verify
    // that the file belongs to the directory we want to use
    // to store log files
    try
    {
        Path javaPath = Paths.get(path);
    } catch (InvalidPathException | NullPointerException ex)
    {
        return false;
    }
    return true;
}

```

Se puede capturar más de un tipo de excepción en un catch

Excepciones definidas por el usuario

- Como las excepciones se organizan en una jerarquía de herencia, para definir nuestras propias excepciones simplemente tenemos que heredar de la clase apropiada
- Si queremos definir un nuevo tipo de excepción verificable podemos heredar de `Exception`
- Si queremos definir una excepción no verificable podemos heredar de `RuntimeException`
- El tipo de la excepción debería ser útil para que el programador entienda la razón de la falla durante el debugging
- Usualmente, definimos nuevas excepciones cuando queremos reportar situaciones de error particulares en nuestra aplicación
 - Y ninguna de las excepciones predefinidas es apropiada



- La Figura muestra un ejemplo de definición de un tipo de excepción verificable
- Para definir una nueva excepción verificable, debemos extender de `Exception`
- En este caso, también redefinimos el constructor para guardar la clave que produjo la falla
- Y redefinimos el método `toString` para que imprima mejor los detalles del error, incluyendo la clave que lo produjo

```
public class NoMatchingDetailsException extends Exception
{
    // The key with no match.
    private String key;

    /**
     * Store the details in error.
     * @param key The key with no match.
     */
    public NoMatchingDetailsException(String key)
    {
        this.key = key;
    }

    /**
     * @return The key in error.
     */
    public String getKey()
    {
        return key;
    }

    /**
     * @return A diagnostic string containing the key in error.
     */
    public String toString()
    {
        return "No details matching: " + key + " were found.";
    }
}
```


Excepciones y subtipos

- Como las excepciones se organizan en una jerarquía de herencia, qué excepciones se capturan en el catch depende del tipo dinámico de la excepción (el tipo del objeto de la excepción que realmente se lanzó)
- El catch capturará todas las excepciones que sean subtipos (subclases) de la excepción declarada
- Por ejemplo, para el método:
- Podemos escribir el siguiente código para capturar ambos tipos de excepciones:
- Esto se debe a que `EOFException` y `FileNotFoundException` son ambas subclases de `Exception`

```
public void process()  
    throws EOFException, FileNotFoundException
```

```
try {  
    ...  
    ref.process();  
    ...  
}  
catch(Exception e) {  
    // Take action appropriate to all exceptions.  
    ...  
}
```

Capturando múltiples excepciones

- O podemos escribir código especializado para capturar y tratar con cada tipo distinto de excepción
- Ver ejemplo del método `process` en la Figura
- En estos casos, en caso de haber excepciones, las cláusulas `catch` se ejecutan en el orden de aparición
- Sólo la primera cláusula `catch` que `matchea` con el tipo de la excepción se ejecuta

```
public void process()  
    throws EOFException, FileNotFoundException
```

```
try {  
    ...  
    ref.process();  
    ...  
}  
catch(EOFException e) {  
    // Take action appropriate to an end-of-file exception.  
    ...  
}  
catch(FileNotFoundException e) {  
    // Take action appropriate to a file-not-found exception.  
    ...  
}
```

Excepciones verificadas vs. no verificadas

- Para elegir qué tipo de excepción usar vamos a seguir las pautas del libro Clean Code [1]
- Para la mayoría de los casos usaremos excepciones no verificadas
- Sólo usaremos excepciones verificadas cuando estas sean críticas y queremos obligar a que sean capturadas por los clientes
 - Ej.: Excepciones en el manejo de archivos, errores de hardware, etc.
- La razón es que las excepciones verificadas fuerzan cambios en el código de los clientes, y esto nunca es una buena idea
 - Si agregamos una excepción verificable en el perfil de un método, tenemos que modificar todos los clientes del método para capturarla o para retornarla al invocante
 - Esto puede producir una cascada de modificaciones, que en proyectos grandes puede afectar a decenas de clases
 - En otras palabras, las excepciones verificadas violan el principio abierto/cerrado
- Notar que las excepciones verificadas no existen en muchos otros lenguajes (C++, C#, Python, Ruby)
- Usar buenas prácticas de diseño es lo más importante para lograr que el software desarrollado sea confiable y robusto

[1] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. 2008.

Aserciones

- Una aserción (`assert`) es una propiedad que debe valer en un punto de la ejecución del programa
 - Si no vale la propiedad, se termina la ejecución y se reporta un error

```
public void removeDetails(String key)
{
    if(key == null) {
        throw new IllegalArgumentException(
            "Null key passed to removeDetails.");
    }
    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";
}
```


Aserciones

- Los `asserts` son muy útiles para revelar distintos tipos de errores de programación
 - Podemos usarlos para verificar precondiciones y postcondiciones
 - En el ejemplo anterior estamos verificando que se cumple parte de la postcondición de `removeDetails`
 - O cualquier propiedad que nos sea de interés
- También sirven como documentación de como debería funcionar el código
- Los `asserts` sólo son evaluados en tiempo de ejecución si se agrega un flag específico en la compilación con Java
- Usualmente, los `asserts` se activan durante el desarrollo, y se deshabilitan cuando el código pasa a producción
 - Esto evita que chequeos potencialmente costosos se ejecuten en el código de producción

Aserciones

- Cuando los chequeos de precondiciones son muy costosos en tiempo, una buena práctica es verificarlos con un `assert` (en lugar de excepciones)
- Por ejemplo:

```
/**
 * @pre 'this' is sorted.
 * @post Returns the index of the first occurrence of
 *        'o' in list 'this'.
 */
public int getIndex(Object o) {
    assert isSorted(this): "Error: The list must be sorted."
    // ... implementation omitted ...
}
```

- De esta manera, nos beneficiamos del chequeo de la precondición durante el desarrollo, pero esta no se ejecuta cuando el código está en producción (lo que haría más lento el programa)

Tests negativos

- Hasta el momento solo hemos escrito tests de "camino feliz" (terminan exitosamente)
- También es muy importante testear que, si un método debería fallar para un input dado, el método realmente falla
 - En caso contrario, el método podría enmascarar la falla, y el error en el código podría pasar desapercibido
 - Estos tests se denominan tests negativos
- Ejemplo: `insertMoney` de `TicketMachine` tiene como precondition que el monto ingresado debe ser positivo
- En el test de la figura se verifica que ejecutar `insertMoney` con `-1` se lanza una excepción
- El test usa la aserción `assertThrows` de JUnit, que es exitosa cuando el método lanza la excepción esperada
 - Y falla en caso contrario

```
/**  
 * @pre 'amount' > 0.  
 * @post Receives an amount of money from a customer.  
 */  
public void insertMoney(int amount)
```

```
@Test  
public void testInsertNegativeMoneyFails()  
{  
    TicketMachine machine = new TicketMachine(10);  
    assertThrows(IllegalArgumentException.class,  
        () -> machine.insertMoney(-1));  
}
```

Tests negativos

- También podemos crear un test negativo para verificar que falla la creación de objetos
- Ej: La creación de una `TicketMachine` falla si el costo de los tickets es `-1`

```
/**
 * @pre 'cost' > 0.
 * @post Create a machine that issues tickets
 *       with a price of 'cost'.
 */
public TicketMachine(int cost)
```

```
@Test
public void testCreateMachineNegativeCostFails()
{
    assertThrows(IllegalArgumentException.class,
        () -> new TicketMachine(-1));
}
```

Programación defensiva

- Como vimos, los errores en el software pueden ser producidos por defectos en los clientes, por problemas de hardware, por fallas en otros programas, por entradas provistas por el usuario, etc.
- La práctica de desarrollar programas que "se defiendan" de los errores se denomina programación defensiva (término inventado por Liskov [2])
- La programación defensiva es una técnica de desarrollo que propone que los programas chequeen la existencia de posibles problemas, y los reporten de manera ordenada
 - Esta práctica permite a la construcción de programas más robustos y confiables
 - Ayuda a revelar fallas en etapas tempranas del desarrollo
 - Facilita el debugging
 - Permite desarrollar programas que se recuperen de manera elegante de las fallas

Programación defensiva

- Para soportar esta técnica, vamos a escribir los programas de manera que:
 - Lancen excepciones ante violaciones de precondiciones
 - O usen `asserts` en caso de que los chequeos sean muy costosos
 - Lancen excepciones ante errores de hardware, información errónea introducida por los usuarios, etc.
 - Y vamos a programar los clientes de manera que puedan recuperarse elegantemente de estos problemas
 - Vamos a escribir tests negativos para verificar que nuestros programas se comportan según lo esperado (fallan) en presencia de fallas

Actividades

- Leer el capítulo 14 del libro "Objects First with Java A Practical Introduction using BlueJ". Sixth Edition. D. Barnes & M. Kölling. Pearson. 2016
- Leer el capítulo 4 del libro "Program Development in Java - Abstraction, Specification, and Object-Oriented Design". B. Liskov & J. Guttag. Addison-Wesley. 2001

Bibliografía

- "Objects First with Java A Practical Introduction using BlueJ". Sixth Edition. D. Barnes & M. Kölling. Pearson. 2016
- "Program Development in Java - Abstraction, Specification, and Object-Oriented Design". B. Liskov & J. Guttag. Addison-Wesley. 2001
- Clean Code: A Handbook of Agile Software Craftsmanship (1st. ed.)". Robert C. Martin. "Prentice Hall. 2008.