

Sesión 22: Excepciones

Programación 2

Ángel Herranz

2021-2022

Universidad Politécnica de Madrid

En capítulos anteriores

- 👍 Tema 1: Intro a POO
- 👍 Tema 2: Clases y Objetos y TADs y módulos
- 👍 Tema 3: Colecciones con *arrays*
- 👍 Tema 4: Cadenas simplemente enlazadas
- 👍 Tema 5: TAD Listas
- 👍 Tema 7: Polimorfismo

En el capítulo de hoy



Tema 8: Excepciones

- Precondiciones y postcondiciones
- Ocultación
- Excepciones en Java



Semántica en javadoc

```
/**  
 * Quita de la lista el elemento en la posicion {@code index}.  
 *  
 * @pre. {@code index} mayor o igual que 0 y menor que {@code size()}  
 * @post. {@code this} después de ejecutar representa una lista como  
 *        {@code this} antes de ejecutar eliminando el elemento que  
 *        ocupaba la posición indicada en el parámetro {@code index}  
 * @param index el índice del elemento que se va a eliminar  
 * @throws RuntimeException si no se cumple la precondition  
 */  
public void remove(int index);
```



Precondición y Postcondición (Def.)

Precondición

Predicado que habla del *estado de mi programa* (incluidos los parámetros de llamada)

Postcondición

Predicado que relaciona el *estado después* de que la operación se haya ejecutado con el *estado antes* de que se ejecute



Precondición y Postcondición (Def.)

Precondición

Predicado que habla del *estado de mi programa* (incluidos los parámetros de llamada)

$$0 \leq index \wedge index < size()$$

Postcondición

Predicado que relaciona el *estado después* de que la operación se haya ejecutado con el *estado antes* de que se ejecute



Precondición y Postcondición (Def.)

Precondición

Predicado que habla del *estado de mi programa* (incluidos los parámetros de llamada)

$$0 \leq \text{index} \wedge \text{index} < \text{size}()$$

Postcondición

Predicado que relaciona el *estado después* de que la operación se haya ejecutado con el *estado antes* de que se ejecute

“*this* despues de ejecutar ...”



La especificación **pre-post**
es un **contrato**

*“Yo (el programador que implementa) **garantizo** que se cumple la postcondición, **siempre y cuando** tú (el programador que usa) **respetes** la precondition.”*

Origen: Lógica de Hoare

- Términos utilizados en la *lógica de Floyd-Hoare* (lógica de la programación, Tony Hoare, 1969)
- Fórmulas de esta forma $\{P\} S \{Q\}$:
Sea S un código: si se cumple P antes de ejecutar S entonces se cumple Q después de ejecutar S
- Ejemplo:

$$\{x = 5\} x = x + 1; \{x = 6\}$$

Las dos partes del contrato

La parte “implementador”

- **Asumes** que nadie va a llamar a tus funciones sin respetar las precondiciones
- Te comprometes a **garantizar** la postcondición (**sólo** si se cumple la precondición)

La parte “cliente” (usa)

- Te comprometes a **respetar** la precondición cuando haces la llamada
- Puedes **asumir que se cumple** la postcondición (**sólo** si se cumplía la precondición)

¿Y si no se cumple la precondition?

- ¿Y si el programador que usa la operación **no respeta** la precondition?

Ej. `lista.get(-2)`

- ¿Qué pasará?

¿Y si no se cumple la precondition?

- ¿Y si el programador que usa la operación **no respeta** la precondition?

Ej. `lista.get(-2)`

- ¿Qué pasará? **Se rompe el contrato:**

¿Y si no se cumple la precondition?

- ¿Y si el programador que usa la operación **no respeta** la precondition?

Ej. `lista.get(-2)`

- ¿Qué pasará? **Se rompe el contrato:**
 1. La postcondición **no está garantizada**
 2. Esto implica que **no se sabe lo que va a pasar**
 3. En el mejor de los casos el programa **se rompe** o eleva una **excepción** (programación defensiva)
 4. En **el peor de los casos los datos se corrompen** (¡no sólo los del programa!)

Design by Contract

Object Oriented Software Construction

Bertrand Meyer

Implementar la clase Vaso para modelizar vasos

```
Vaso(double capacidad)
```

```
void llenar(double cantidad)
```

```
void vaciar(double cantidad)
```

1. Clase vacía compilable sólo con documentación¹
- 2a. Terminar la implementación
- 2b. Mientras, yo escribo un programa de prueba

¹ *Javadoc* precondiciones y postcondiciones

Pensad en un diálogo

- Yo: “Necesito métodos para comprobar la *precondición*, no vale que me exijas cumplir algo que no puedo comprobar”
- Tú: ¿Qué métodos necesitas?
- Yo: “¿Qué va a pasar cuando una *PRE* no se cumpla?”
- Tú: Mmmm. Deja que piense. . .

Documentar primero: precondition

```
/**  
 * Anade anade una cantidad al vaso  
 *  
 * @pre. el contenido del vaso más {@code cantidad}  
 * no puede superar la capacidad del vaso  
 * @post. el contenido del vaso se habrá incrementado en  
 * {@code cantidad}  
 *  
 * @param cantidad Mililitros a anadir al vaso  
 */  
public void llenar(double cantidad)  
{  
    ...  
}
```

Method Detail

llenar

```
public void llenar(double cantidad)
    throws CapacidadSuperada
```

Añade añade una cantidad al vaso

PRE: el contenido del vaso más cantidad no puede superar la capacidad del vaso

POST: el contenido del vaso se habrá incrementado en cantidad

Parameters:

cantidad - Mililitros a añadir al vaso

Throws:

CapacidadSuperada

Métodos para comprobar la *PRE*

```
/**  
 * Dice qué capacidad tiene el vaso  
 * @return capacidad del vaso  
 */  
public double capacidad() {  
    ...  
}
```

```
/**  
 * Dice qué contenido tiene el vaso  
 * @return contenido del vaso  
 */  
public double contenido() {  
    ...  
}
```

Más formalmente

```
/**
 * Anade anade una cantidad al vaso
 *
 * @pre. {@code contenido() + cantidad <= capacidad()}
 * @post. el contenido del vaso se habrá incrementado en
 * {@code cantidad}
 *
 * @param cantidad Mililitros a anadir al vaso
 */
public void llenar(double cantidad)
{
    ...
}
```

Documentar primero: invariante

```
/**
 * Las instancias de Vaso representan vasos de la realidad,
 * que se pueden llenar y vaciar.
 *
 * @inv. {@code
 *   this.contenido() <= this.capacidad()
 *   && this.contenido() >= 0
 *   && this.capacidad() > 0
 * }
 */
public class Vaso {
  ...
}
```

¿Para qué escribir invariantes?

- Las invariantes² ayudan a entender el código
- Se puede asumir que **se cumplen antes** de ejecutar un método
- Están **garantizados después** de ejecutar un método

²*Propiedades invariantes*

¿Demasiado silencioso?

```
Vaso v = new Vaso(200);  
v.llenar(250);
```

¿Y nada se *rompe*?

No hacer nada

No hacer nada

```
public void llenar(double cantidad)
{
    contenido += cantidad;
}
```

El vaso se *desborda*

Opción 2:

Hacer *algo*

Opción 2: programación defensiva

Hacer *algo*

```
public void llenar(double cantidad)
{
    if (contenido + cantidad > capacidad) {
        // Hacer algo "defensivo"
    }
    contenido += cantidad;
}
```

Romper el programa

Romper el programa

```
public void llenar(double cantidad) {  
    if (contenido + cantidad > capacidad) {  
        System.err.println("Capacidad superada");  
        System.exit(1);  
    }  
    contenido += cantidad;  
}
```

Lanzar³ una excepción



Me gusta

³*to throw* – lanzar, disparar, elevar (*raise*), etc.

Lanzar³ una excepción

```
public void llenar(double cantidad) {  
    if (contenido + cantidad > capacidad)  
        throw new RuntimeException("Capacidad superada");  
    contenido += cantidad;  
}
```

- **throw** se comporta parecido a un **return** pero rompiendo el flujo de ejecución

³to *throw* – lanzar, disparar, elevar (*raise*), etc.

Excepciones en Java

- Objetos (¡**new**!) de clases especiales:
 `RuntimeException extends Exception`
- Buenas prácticas, por ejemplo:
- Una clase por cada tipo de situación excepcional
- Al llenar: excepción por `capacidad superada`

Excepciones en Java

- Objetos (¡new!) de clases especiales:
 `RuntimeException extends Exception`
- Buenas prácticas, por ejemplo:
- Una clase por cada tipo de situación excepcional
- Al llenar: excepción por `capacidad superada`
- Al vaciar: excepción por `contenido insuficiente`

Excepciones en Java

- Objetos (¡new!) de clases especiales:
 `RuntimeException` **extends** `Exception`
- Buenas prácticas, por ejemplo:
- Una clase por cada tipo de situación excepcional
- Al llenar: excepción por `capacidad superada`
- Al vaciar: excepción por `contenido insuficiente`
- Al construir: excepción por `capacidad no positiva`
- Al llenar o vaciar: excepción por `contenido no positivo`

Definir el tipo de excepción

```
public class CapacidadSuperada  
  
{  
    public CapacidadSuperada() {  
    }  
}
```

Definir el tipo de excepción

```
public class CapacidadSuperada
    extends Exception
{
    public CapacidadSuperada() {
    }
}
```

```
public void llenar(double cantidad)

{

    contenido += cantidad;
}
```

```
public void llenar(double cantidad)

{
    if (contenido + cantidad > capacidad) {
        throw new CapacidadSuperada();
    }
    contenido += cantidad;
}
```

```
public void llenar(double cantidad)
```

```
{
```

```
    if (contenido + cantidad > capacidad) {
```

```
        throw new CapacidadSuperada();
```

```
    }
```

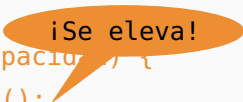
```
    contenido += cantidad;
```

```
}
```

¡Se eleva!

```
public void llenar(double cantidad)
```

```
{  
    if (contenido + cantidad > capacidad) {  
        throw new CapacidadSuperada();  
    }  
    contenido += cantidad;  
}
```

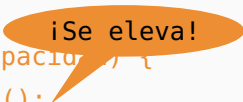


```
$ javac Vaso.java
```

```
Vaso.java:55: error: unreported exception CapacidadSuperada;  
    must be caught or declared to be thrown  
        throw new CapacidadSuperada();  
        ^
```

```
$
```

```
public void llenar(double cantidad)
    throws CapacidadSuperada
{
    if (contenido + cantidad > capacidad) {
        throw new CapacidadSuperada();
    }
    contenido += cantidad;
}
```



Elevar y declarar la excepción

```
public void llenar(double cantidad)
    throws CapacidadSuperada
{
    if (contenido + cantidad > capacidad) {
        throw new CapacidadSuperada();
    }
    contenido += cantidad;
}
```

¡Declaración!

¡Se eleva!

Capturar la excepción

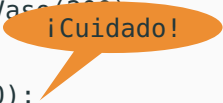
```
Vaso v = new Vaso(200);
```

```
v.llenar(100);
```

```
System.out.println(v);
```

Capturar la excepción

```
Vaso v = new Vaso(200);
```



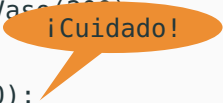
¡Cuidado!

```
v.llenar(100);
```

```
System.out.println(v);
```

Capturar la excepción

```
Vaso v = new Vaso(200);
```



¡Cuidado!

```
v.llenar(100);
```

```
System.out.println(v);
```

```
$ javac PruebaVaso.java
```

```
PruebaVaso.java:5: error: unreported exception CapacidadSuperada;  
must be caught or declared to be thrown
```

```
    v.llenar(100);
```

Capturar la excepción

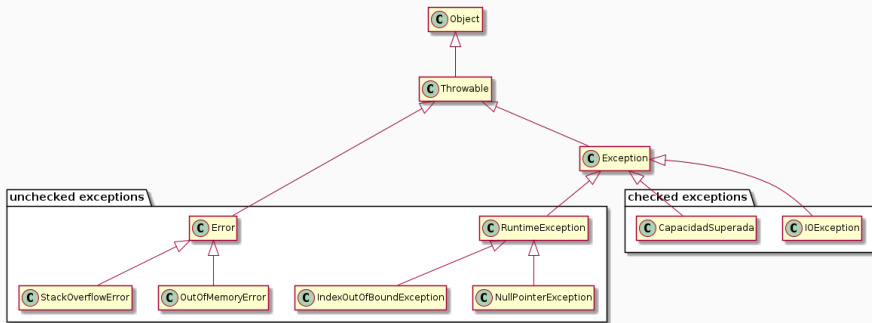
```
Vaso v = new Vaso(200);  
try {  
    v.llenar(100);  
    System.out.println(v);  
}  
catch (CapacidadSuperada excepcion) {  
    System.err.println("Prueba incorrecta");  
    System.err.println("Se ha superado la capacidad");  
    System.exit(1);  
}
```



- Crear la excepción `ContenidoInsuficiente` para vaciar
- Modificar los tests

Jerarquía de excepciones

- *checked*: capturar o elevar, el compilador avisa
- *unchecked*: el compilador no avisa



Aprovechando la herencia

```
Vaso v = new Vaso(100.0);  
try {  
    ...  
    v.llenar(...);  
    ...  
    v.vaciar(...);  
    System.out.println(v);  
}
```

Aprovechando la herencia

```
Vaso v = new Vaso(100.0);  
try {  
    ...  
    v.llenar(...);  
    ...  
    v.vaciar(...);  
    System.out.println(v);  
}  
catch (Exception e) {  
    // CapacidadSuperada y ContenidoInsuficiente heredan de Exception  
    // y cualquiera de las dos se capturan en este bloque  
    System.err.println("Error al llenar o vaciar el vaso");  
    System.exit(1);  
}
```

Excepciones con detalles del problema

```
public class CapacidadSuperada extends Exception
{
    private double exceso;
    public PilaAcotadaLlena(double exceso) {
        this.exceso = exceso;
    }
    public exceso() {
        return exceso;
    }
}
```

Algunos métodos en Throwable

- `String getMessage()`
Returns the detail message string of this throwable
- `StackTraceElement[] getStackTrace()`
Provides programmatic access to the stack trace information printed by `printStackTrace()`
- **`void printStackTrace(PrintStream s)`**
Prints this throwable and its backtrace to the standard error stream

Inspeccionando la excepción

```
Vaso v = new Vaso(100.0);  
try {  
    ...  
    v.llenar(...);  
    ...  
    v.vaciar(...);  
    System.out.println(v);  
}  
catch (Exception e) {  
    System.err.print("Se ha detectado un error: ");  
    System.err.println(e.getMessage());  
    e.printStackTrace(System.err);  
    System.exit(1);  
}
```

Finalizando i

- En general, cada excepción puede **merecer un tratamiento diferente**
- Pero quizás parte del **tratamiento** sea **común**
- O incluso queremos un **tratamiento para el resto de excepciones** que no sepamos como tratar
- Para ello Java permite añadir **finally**

Finalizando ii

```
Vaso v = new Vaso(100.0);  
try {  
    ...  
    v.llenar(...);  
    ...  
    v.vaciar(...);  
    System.out.println(v);  
}  
catch (CapacidadSuperada capSup) {  
    System.err.println("Error al llenar el vaso");  
}  
catch (ContenidoInsuficiente contInsuf) {  
    System.err.println("Error al vaciar el vaso");  
}
```

Finalizando ii

```
Vaso v = new Vaso(100.0);
try {
    ...
    v.llenar(...);
    ...
    v.vaciar(...);
    System.out.println(v);
}
catch (CapacidadSuperada capSup) {
    System.err.println("Error al llenar el vaso");
}
catch (ContenidoInsuficiente contInsuf) {
    System.err.println("Error al vaciar el vaso");
}
finally {
    // El bloque "finally" se ejecuta siempre que hay una excepción!
    System.err.println("Revisa tus tests");
    System.exit(1);
}
```