

Sesión 17: Excepciones

Programación 2







6. Excepciones

Ángel Herranz

2019-2020

Universidad Politécnica de Madrid

En capítulos anteriores

-  Tema 1: Clases y Objetos
-  Tema 2: Colecciones acotadas de Objetos
-  Tema 4: Tipos Abstractos de Datos
-  Tema 3: Programación Modular
-  Tema 5: Herencia y Polimorfismo
-  Tema 7: Implementación de TADs lineales

Nodo<T>

En el capítulo de hoy



Tema 6: Excepciones

- Precondiciones y postcondiciones
- Ocultación
- Excepciones en Java

Precondiciones y postcondiciones

¿Qué sabéis?

Precondiciones y postcondiciones

¿Qué sabéis?

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

Precondición

Propiedad que se **asume antes**
de ejecutar un código

Propiedad

Propiedad que se **garantiza**
después de ejecutar un código

Dos roles

Cuando lo implementas

- **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)

Cuando lo usas

- **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 cumplo con la precondition?

Cuando lo implementas

- Puedes hacer lo que te de la gana

Cuando lo usas

- No sabes lo que puede pasar

¿Y si no cumplo con la precondition?



Cuando lo implementas

- Puedes hacer lo que te de la gana
- Incluido... ¡formatear el disco duro!

Cuando lo usas

- No sabes lo que puede pasar
- ¡A lo mejor se formatea el disco duro!

¿A qué suena esto?

Design by Contract

¿A qué suena esto?

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* documentar precondiciones y postcondiciones

Paso a paso

- “Necesito **métodos para comprobar la precondition**, no vale que me exijas cumplir algo que no puedo comprobar”
- ¿**Qué métodos** necesitas?
- ¿No te parece que el incumplimiento de las *PREs* es **demasiado silencioso**?
- Una opción es que `llenar` devuelva un booleano que diga si se ha derramado
- A esto se le llama **programación defensiva** y en general, **no es buena idea**
- ¿Otras opciones?

Documentar primero: precondition

```
/**  
 * Anade anade una cantidad al vaso  
 *  
 * <br><strong>PRE:</strong>  
 * el contenido del vaso más <code>cantidad</code> no puede  
 * superar la capacidad del vaso  
 * <br><strong>POST:</strong>  
 * el contenido del vaso se habrá incrementado en  
 * <code>cantidad</code>  
 *  
 * @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
 *
 * <br><strong>PRE:</strong>
 * <code>contenido() + cantidad <= capacidad()</code>
 * <br><strong>POST:</strong>
 * el contenido del vaso se habrá incrementado en
 * <code>cantidad</code>
 *
 * @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.
 *
 * <br>
 * <strong>INV:</strong>
 * <code>
 *   this.contenido() <= this.capacidad()
 *   && this.contenido() >= 0
 *   && this.cantidad() > 0
 * </code>
 */
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
- Los invariantes suelen implicar precondiciones

²*Propiedades invariantes*

¿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
- Los invariantes suelen implicar precondiciones

¿Se te ocurre algún ejemplo?

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

- Es como romper el programa pero dando la oportunidad de recuperarse al que usa el código

Me gusta

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

Excepciones en Java

- Objetos de clases especiales:
extends Exception
- Una clase por cada tipo de situación excepcional
- Al llenar: excepción por **capacidad superada**

Excepciones en Java

- Objetos de clases especiales:
extends Exception
- Una clase por cada tipo de situación excepcional
- Al llenar: excepción por **capacidad superada**
- Al vaciar: excepción por **contenido insuficiente**

Excepciones en Java

- Objetos de clases especiales:
extends Exception
- 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!

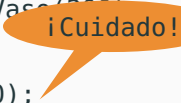
Capturar la excepción

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

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

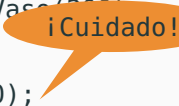
Capturar la excepción

```
Vaso v = new Vaso(200);  
v.llenar(100);
```



Capturar la excepción

```
Vaso v = new Vaso(200);  
v.llenar(100);
```



```
$ 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);  
}  
catch (CapacidadSuperada excepcion) {  
    System.err.println("Prueba incorrecta");  
    System.err.println("Se ha superado la capacidad");  
    System.exit(1);  
}
```