

0101010
0100101
1101010

UD9.2- GESTIÓ D'EXCEPCIONS - Llançament i definició

Programació –1er DAW/DAM

0. CONTINGUTS

- Llançament d'excepcions
- La clàusula throws
- Creació d'excepcions pròpies

1. Llançament d'excepcions

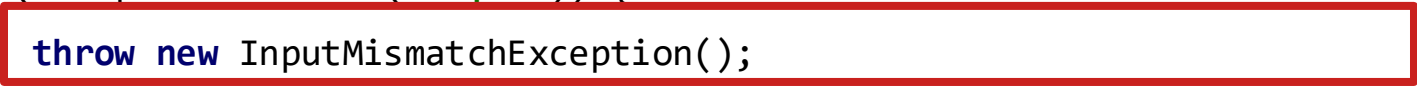

- Fins ara, s'han **capturat les excepcions** que genera **automàticament la JVM** o els **mètodes** als que accedim.
- Els **programadors** també **poden llançar excepcions** com a resposta a possibles **situacions anòmales**.
 - Es fa ús de la sentència **throw new** `Exception()` que permet llançar una `Exception`; ja siga **qualsevol excepció existent** o una creada per nosaltres mateixos.

throw new nomDeLaExcepcio();

1. Llançament d'excepcions

- La instrucció **throw** provoca que s'abandoni l'execució del mètode on es troba i es passe el **control** al mètode **invocador**, al catch que captura l'error.

```
public static void main(String[] args) {  
    try {  
        askContinue();  
    } catch (InputMismatchException e) {  
        System.out.println("L'error s'ha produït a causa de " + e.getMessage());  
    }  
}  
  
public static boolean askContinue(){  
    Scanner scanner = new Scanner(System.in);  
    System.out.println("Voleu Continuar (Si/No)");  
    String response = scanner.next();  
    if (!response.matches("Si|No")) {  
        throw new InputMismatchException();  
    }  
    return response.equals("Si");  
}
```



1. Llançament d'excepcions

- Els constructors de la classe `Exception` permeten indicar un **missatge descriptiu** sobre l'error que ha provocat la Excepció.
- Aquest missatge podrem recuperar-lo al manejador a través del mètode `getMessage()`

```
public static void main(String[] args) {  
    try {  
        askContinue();  
    } catch (InputMismatchException e) {  
        System.out.println("L'error s'ha produït a causa de "+e.getMessage());  
    }  
}  
  
public static boolean askContinue(){  
    Scanner scanner = new Scanner(System.in);  
    System.out.println("Voleu Continuar (Si/No)");  
    String response = scanner.next();  
    if (!response.matches("Si|No")) {  
        throw new InputMismatchException("El format de la data ha de ser Sí/No");  
    }  
    return response.equals("Si");  
}
```

1.1 Activitat prèvia

Activitat 7. Crea un mètode `getEdat()` que demane a l'usuari un número entre 10 i 50. Si el número no es troba al rang, haurà de llançar una excepció de tipus `InputMismatchException` amb un missatge descriptiu de l'error que s'ha produït.


Captura des del `main()` l'excepció anterior i mostra el missatge.

Prova el mètode amb els valors 10, 20, 50, 200 i 9.

1.2 Directrius per capturar excepcions

- **No registres excepcions per a després llançar-les (rellançament):** *Estaries duplicant el tractament i els possibles missatges d'error.*

```
try {  
    new Long("pepe");  
} catch (NumberFormatException e) {  
    log.error(e);  
    throw e;  
}
```



2. La clàusula throws(I)

- Les **excepcions explícites** (aquelles que no hereten de `RuntimeException`) representen una **situació anòmla** al programa (**no un error**) que deu **tractar-se obligatòriament**.

```
public class TestExcepcionExplicita {

    public static void main(String[] args) {
        generarExcepcio();
    }

    public static void generarExcepcio() {
        throw new Exception("Excepció explícita");
    }

}
```

Produeix un error de compilació,
l'excepció no ha sigut capturada

```
public class TestExcepcionExplicita {

    public static void main(String[] args) {
        generarExcepcio();
    }

    public static void generarExcepcio() {
        try {
            throw new Exception("Excepció explícita");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```

En aquest exemple, la
captura es produeix en
el mateix mètode on
l'excepció és llançada

2. La clàusula `throws`(II)

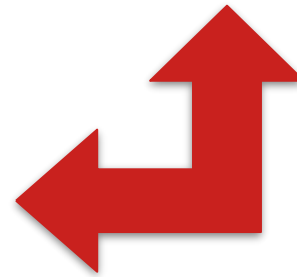
- Si un mètode pot llançar excepcions explícites i no va a manejar-les en el mateix mètode on és llançada (propagació) ho **ha d'indicar** als seus invocadors mitjançant la clàusula `throws`.

```
public boolean nomMètode() throws Exception, NomExcepcion2,... {
```

```
    throw new Exception;
```

```
    throw new NomExcepcion2;
```

```
}
```



2. La clàusula `throws`(III)

- El **mètode invocador**, haurà **obligatòriament** de dur a terme el seu **tractament** o bé indicar que vol propagar-ho a través de la sentència `throws`

```
public class TestPropagacionExcepcionExplicita {  
  
    public static void main(String[] args) {  
        try {  
            generarExcepcio();  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
  
    public static void generarExcepcio() throws Exception {  
  
        throw new Exception("Excepció explícita");  
  
    }  
}
```

2.1 La clàusula `throws`. Exemple

```
public class PersonaAdulta {  
  
    1 usage  
    protected String nombre;  
  
    1 usage  
    protected int edad;  
  
    protected boolean casado;  
  
    public PersonaAdulta(String nombre, int edad) {  
        try {  
            if (edad < 10) {  
                throw new InvalidObjectException("La persona debe ser adulta");  
            }  
        } catch (InvalidObjectException e) {  
            throw new RuntimeException(e);  
        }  
        this.nombre = nombre;  
        this.edad = edad;  
    }  
}
```

1
sense sentència
`throws`

L'excepció es llança a un
mètode i es captura dins d'ell
mateix **(SENSE SENTIT)**



2.1 La clàusula `throws`. Exemple

```

public class PersonaAdulta {
    2 usages
    private String nombre;
    2 usages
    private int edad;

    2 usages
    public PersonaAdulta(String nombre, int edad) throws InvalidObjectException {
        this.nombre = nombre;
        if (edad < 18)
            throw new InvalidObjectException("No es adulta la persona " + nombre + " porque tiene " + edad + " años.");
        this.edad = edad;
    }

    @Override
    public String toString() {
        return String.format("%s - %s", nombre, edad);
    }

    no usages
    public static void main(String[] ar) {
        try {
            System.out.println(new PersonaAdulta( nombre: "Ana", edad: 50));
            System.out.println(new PersonaAdulta( nombre: "Juan", edad: 13));
        } catch (InvalidObjectException ex) {
            System.out.println(ex.getMessage());
        }
    }
}

```

2
amb sentència
`throws`

L'excepció es llança a un
mètode i es captura (en
aquest cas) al mètode que
l'invoca

Activitat Prèvia

Activitat 8. Crea un programa anomenat `WaitApp` amb una funció anomenada `waitSeconds(int seconds)` que rebrà un nombre de segons (sencer) com a paràmetre.

Aquesta funció haurà de cridar al mètode `Thread.sleep` per pausar el programa la quantitat de segons passats com a paràmetre (*aquest mètode funciona amb mil·lisegons, per la qual cosa caldrà convertir els segons rebuts a mil·lisegons*).

Nota: El mètode `sleep` pot produir una excepció `InterruptedException`, en aquest cas volem manejar-la des del mètode `main()` (que cridarà a la funció `waitSeconds`).

Després d'esperar els segons especificats, el programa ha de mostrar un missatge de finalització.

De quin subtipus (**implícita** o **explícita**) és l'excepció que estem capturant?

3. Definició d'excepcions pròpies

- Les **excepcions de Java** controlen els errors més habituals.
- La potència d'aquest mecanisme radica en la capacitat de **crear excepcions pròpies** per controlar **errors de la nostra aplicació**.
- La creació d'una subclasse és molt senzilla, només cal definir una subclasse de `Exception` o `RuntimeException` depenent de si volem que siga tractada obligatòriament o no.

```
class MiErrorException extends Exception {  
    ...  
}
```

← excepció **explícita**

```
class MiErrorException extends RuntimeException {  
    ...  
}
```

← excepció **implícita**

3.1 Exemple

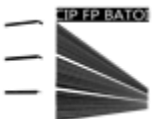
- La següent classe representa una regleta amb un màxim **10 endolls**
- Fins ara havíem definit que la funció connect() tornava un **booleà** indicant si l'aparell ha **pogut ser engegat o no** (programació defensiva)

```
public class Plug {  
  
    private Connectable[] connectables;  
  
    private int MAX_CONNECTABLE = 10;  
  
    public Plug() {  
        this.connectables = new Connectable[MAX_CONNECTABLE];  
    }  
  
    public boolean connect(Connectable device) {  
        for(int i = 0; i < connectables.length; i++) {  
            if (connectables[i] == null) {  
                connectables[i] = device;  
                device.turnOn();  
                return true;  
            }  
        }  
        return false;  
    }  
}
```

3.1 Exemple

- Podem definir una **excepció pròpia** RegletaPlenaException que herete de Exception i llançar-la quan intentem connectar un aparell electrònic i no hi ha espai.

```
public class RegletaPlenaException extends Exception {  
    public RegletaPlenaException() {  
        super("No hi ha endolls lliures"); //establim un missatge per defecte  
    }  
}  
  
public void connect(Connectable device) throws RegletaPlenaException {  
    for (int i = 0; i < connectables.length; i++) {  
        if (connectables[i] == null) {  
            connectables[i] = device;  
            device.turnOn();  
            return;  
        }  
    }  
    throw new RegletaPlenaException();  
}
```



Activitat prèvia

Activitat 9.- Es vol crear una classe que represente un tanc per transportar líquids (aigua, gasolina, etc.). Cal implementar els mètodes per afegir i retirar líquid del tanc. Si s'excedeix la capacitat del tanc o s'intenta traure aigua quan està buit, cal llançar l'excepció `TancPleException` o `TancBuitException` respectivament.

Escriu un programa `TestTanc` que ens permeti validar el funcionament.

Tanque
- capacidad: int - carga: int
+ Tanque(capacidad: int) + agregarCarga(cantidad: int) :void + retirarCarga(cantidad: int): void

Fes 2 versions, una amb excepcions explícites i altra amb implícites.
Per a esta última implementa mètodes que ens permetran saber si el tanc està ple o no

Això és tot... de moment :-)