CLASE 6 - Exceptions

Excepciones – 2022

- **Excepciones.**
- **Buenas Prácticas.**
- **Por qué usar Excepciones?**
- **Qué es una excepción?**
- **Tipos de excepciones.**
- **Definiciones**
- **Esquema básico del uso de excepciones. throw, throws, try, catch y finally.**
- **Excepciones más específicas.**
- **Qué información debe almacenar la excepción.**
- **Lanzamiento de excepciones**
- **3 posibilidades cuando un método lanza una excepción.**
- **Throws y la redefinición de métodos .**
- **Uso correcto de finally**
- **Cuándo y cómo usar excepciones.**
- **Creando nuevas excepciones**

CLASE 6 - Exceptions

En el mundo de la programación hay algo que siempre ocurre: los **errores** en los programas.

Pero ¿qué sucede realmente después de que ha ocurrido el error? ¿Qué podemos hacer con él? ¿Cómo nos damos cuenta que se ha producido?

El lenguaje Java utiliza excepciones para permitir trabajar mejor con los errores.

Excepciones

Qué sucede cuando un método se aplica a un conjunto de datos que no pertenecen estrictamente al dominio permitido?

Por ejemplo, en el método dividir:

```
public int dividir(int a, int b)
{
     return a/b;
}
```

Se invoca el método con los argumentos:

```
int rta = dividir (120, 0);
```

QUÉ HAGO !!!

Excepciones

Qué sucede cuando un método se aplica a un conjunto de datos que no pertenecen estrictamente al dominio permitido?

Por ejemplo, en el método dividir:

```
public int dividir(int a, int b)
{
    return a/b;
}
```

Se invoca el método con los argumentos:

```
int rta = dividir (120, 0);
```

Excepciones

Qué sucede cuando un método se aplica a un conjunto de datos que no pertenecen estrictamente al dominio permitido?

Por ejemplo, en el método dividir:

```
public int dividir(int a, int b)
{
    return a/b;
}
```

Se invoca el método con los argumentos:

```
int rta = dividir (120, 0);
```

Excepciones – buenas prácticas

- El problema no se puede solucionar dentro del ámbito del método
- No conviene delegar la responsabilidad de verificar la entrada de datos al resto del programa (únicamente) para que el método no provoque un error.
- Debe mantenerse la robustez del método y no retornar resultados “tipo bandera” que indiquen invalidez.
- La verificación no debe ser parte del flujo normal de código (del método)
- La solución debe delegarse a otra zona del programa.

Excepciones – por qué usarlas?

Mediante el uso de excepciones para controlar errores, los programas Java tienen las siguientes ventajas frente a las técnicas de manejo de errores tradicionales.

Separar el Manejo de Errores del Código "Normal“

Estará en una zona separada donde podremos tratar las excepciones como un código ‘especial’.

Propagar los Errores sobre la Pila de Llamadas

Podemos propagar el error al primer método que llamó a los diversos métodos hasta que llegamos al error.

Agrupar Errores y Diferenciación

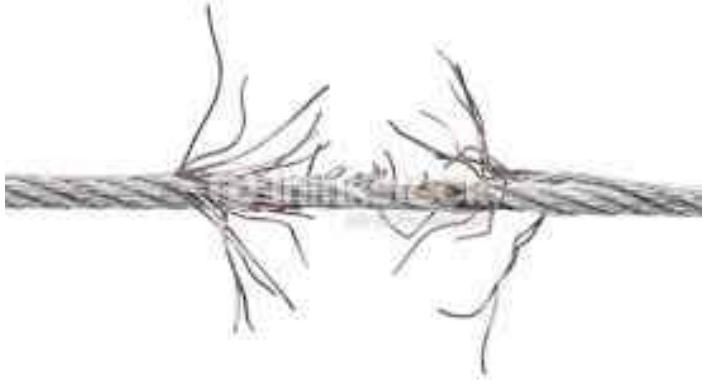
Gracias a esto tenemos todos los posibles errores juntos y podemos pensar una manera de tratarlos que sea adecuado.

¿Qué es una excepción?

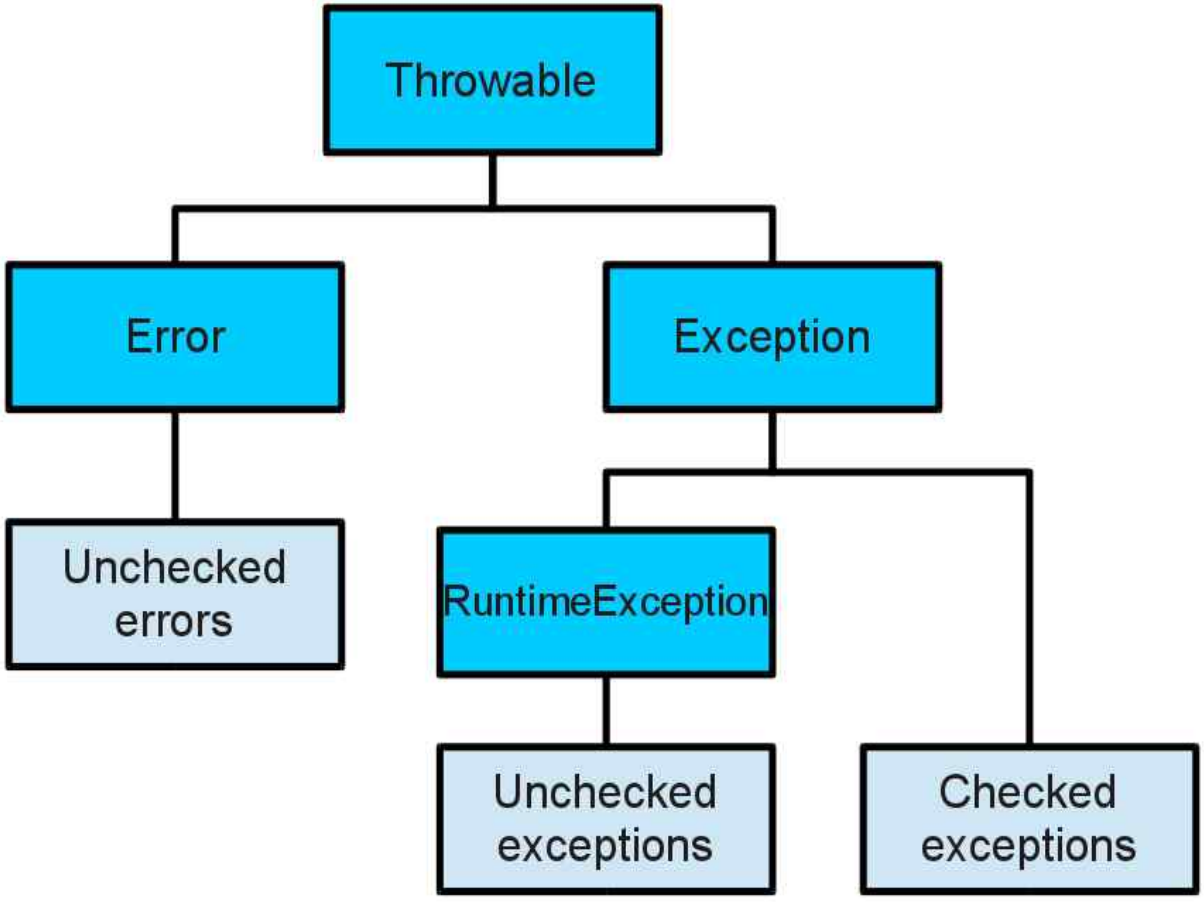
Una excepción es un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias. O sea, algo que altera la ejecución normal.

Muchas clases de errores pueden generar excepciones -- desde problemas de hardware, como la avería de un disco duro, a los simples errores de programación, como tratar de acceder a un elemento de un array fuera de sus límites.

```
X = A / 0;
```



Las clases involucradas – Jerarquía de excepciones

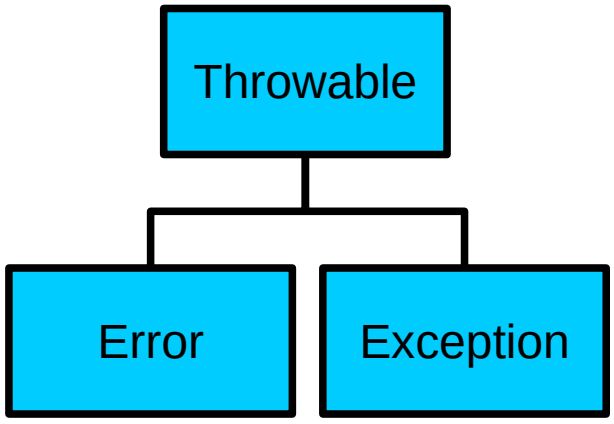


Las excepciones son objetos.

Todos los tipos de excepción (las clases) deben extender de la clase **Throwable** o de una de sus subclases.

Por convenio, los nuevos tipos de excepción extienden a **Exception,** una subclase de Throwable

Las clases involucradas – Jerarquía de excepciones



Throwable

Clase base que representa todo lo que se puede “lanzar” en Java

Error

Subclase de Throwable que indica problemas graves que una aplicación no debería intentar solucionar.
Ejemplos: Memoria agotada, error interno de la JVM...

Exception

Exception y sus subclases indican situaciones que una aplicación debería tratar de forma razonable.
Los dos tipos principales de excepciones son:

- RuntimeException (errores del programador, como una división por cero o el acceso fuera de los límites de un array)
- IOException (errores que no puede evitar el programador, generalmente relacionados con la entrada/salida del programa).

Definiciones

Comprobar errores en forma clara

Las excepciones proporcionan una forma clara de comprobar posibles errores sin oscurecer el código.

Las condiciones de error forman parte del contrato

Convierten las condiciones de error que un método puede señalar en una parte explícita del contrato del método (lista de excepciones).

en forma de lista de excepciones

La lista de excepciones pueden ser vistas por el programador, comprobada por el compilador y preservada (si es necesario) por las clases extendidas que redefinan el método.

Definiciones

Espacio diferente de solución

Al producirse una excepción, se detiene el flujo de ejecución normal (esperado). La ejecución continúa en otro espacio, especialmente diseñado para el tratamiento de esa excepción.

Código limpio

De esta forma, no se oscurece el código específico de cada método, con instrucciones que se ocupen de solucionar estos problemas.

Excepciones: esquema básico de la solución

```
Metodo(...) throws Zona de propagación del error
{
    try
    {
        Zona de ejecución normal
    }
    catch( Exception e)
    {
        Zona de recuperación
    }
    finally
    {
        Zona de finalización
    }
```

Excepciones: esquema básico de la solución

```
Metodo(...)
{
```

throws
Zona de propagación del error

```
try
{
        Zona de ejecución normal
}
catch( Exception e)
{
        Zona de recuperación
}
finally
{
        Zona de finalización
}
```

```
}
```

Dentro de la zona denominada **`try`**, se establecen las instrucciones que conforman el código “normal”, el que determina la funcionalidad de método.
Si alguna de estas instrucciones provoca un problema (excepción), no se continúa ejecutando ninguna de las siguientes instrucciones del bloque **`try`** y se provoca el lanzamiento de una excepción (creación de un objeto de tipo Exception y “lanzamiento”).

Excepciones: esquema básico de la solución

```
Metodo(...)
{
```

throws
Zona de propagación del error

try
{
Zona de ejecución normal
}

catch(**Exception e**)
{
Zona de recuperación
}

finally
{
Zona de finalización
}

```
}
```

El lanzamiento de un objeto de tipo Exception altera el flujo normal del programa.

La zona **`catch`** se encarga de recibir por parámetro dicho objeto y en base a los datos aportados por el mismo, se elabora una estrategia de solución al problema ocurrido.

Excepciones: esquema básico de la solución

```
Metodo(...)
{
```

throws
Zona de propagación del error

try
{
Zona de ejecución normal
}

catch(**Exception e**)
{
Zona de recuperación
}

finally
{
Zona de finalización
}

```
}
```

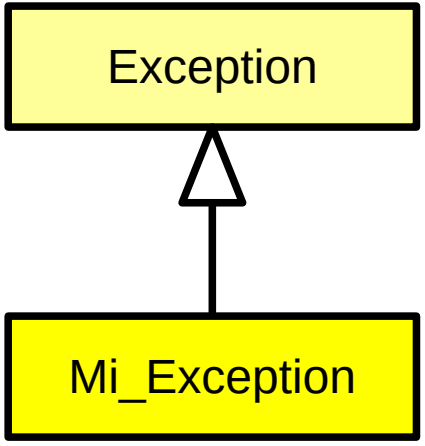
Si el cuerpo del bloque try llega a comenzar su ejecución, el bloque finally siempre se ejecutará...

- Detrás del bloque try si no se producen excepciones
- Después de un bloque catch si éste captura la excepción.
- Justo después de que se produzca la excepción si ninguna cláusula catch captura la excepción y antes de que la excepción se "***propague hacia arriba".***

Excepciones: Profundizando un poco más

Excepciones más específicas

Podemos crear nuestras propias excepciones, como una forma de separar cada tipo de problemática.




```
Metodo(...)
{
```

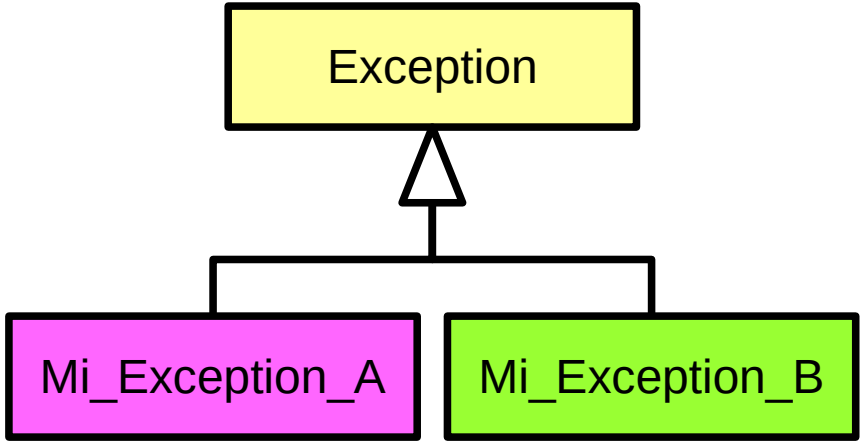
throws
Zona de propagación del error

```
try
{
        Zona de ejecución normal
}
catch( Mi_Exception e)
{
        Zona de recuperación
        Para un tipo de excepción
}
finally
{
        Zona de finalización
}
```

Excepciones: Profundizando un poco más

Excepciones más específicas

Podemos crear nuestras propias excepciones, como una forma de separar cada tipo de problemática.
Java trae un gran conjunto de excepciones específicas.




```
Metodo(...)
{
```

throws
Zona de propagación del error

try

```
catch( Mi_Exception_A  a)
{...}
catch( Mi_Exception_B  b)
{...}

//todos los catch necesarios
```

finally

Excepciones: Profundizando un poco más

Excepciones más específicas

Cada zona **`catch`** se encarga de atrapar la excepción de su propio tipo.

De esta forma, se organizan los espacios de recuperación para cada tipo de excepción.

La excepción lanzada (una sola) "revisa" cada **`catch`** y es atrapada por el primer **`catch`** que concuerda con su tipo (o de su padre...)

Debe cuidarse el órden en que se ubican los **`catch`**, primero deben estar las excepciones hijas y luego las padre (pensar).

```
Metodo(...)
{
```

`throws`
Zona de propagación del error

try

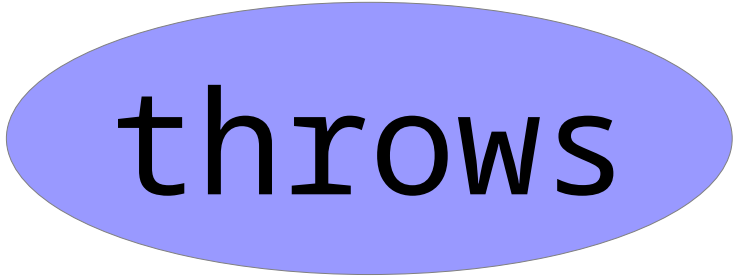
```
catch( Mi_Exception_A  a)
{...}
catch( Mi_Exception_B  b)
{...}

//todos los catch necesarios
```

finally

Excepciones: Profundizando un poco más

Qué sucede si se lanza una excepción que no está contemplada en ningún catch?

→ Se propaga




```
Metodo(...)
{
```

throws
Zona de propagación del error

try

```
catch( Mi_Exception_A  a)
{...}
catch( Mi_Exception_B  b)
{...}

//todos los catch necesarios
```

finally

Excepciones: Profundizando un poco más

Throws

Es la zona de propagación de errores.

Se utiliza para delegar la solución del problema al método invocante (el que pide la ejecución del método actual).

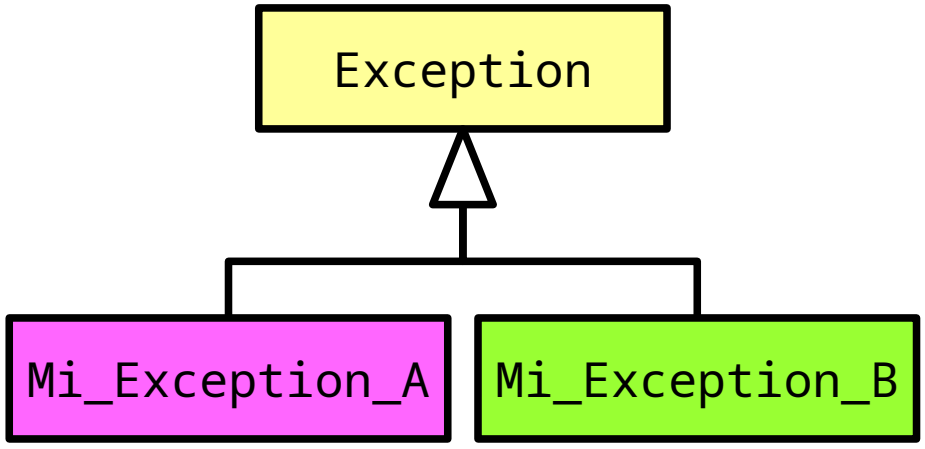
Se activa cuando no existe un **`catch`** en el cuerpo del método que provoca la excepción que concuerde con el tipo de la excepción.

En esta zona, se enumeran todas las excepciones que serán propagadas por el método.

```
metodo1(...) throws
{
```

```
try
catch( Mi_Exception_A  a)
{...}
catch( Mi_Exception_B  b)
{...}

//todos los catch necesarios
finally
```

Excepciones: Profundizando un poco más



Dentro de la zona try, una instrucción lanza una excepción de tipo **Mi_Exception_B.**

```
metodo1(...) throws Mi_Exception_B
{
    try
    {
        alguna instrucción provoca
        el lanzamiento de una excepción
        de tipo Mi_Exception_B
    }

    catch( Mi_Exception_ A  a)
    {
        Zona de recuperación de Mi_Exception_A
    }
    finally
```

Excepciones: Profundizando un poco más

```
metodo2
{
    try
    {    …..
         metodo1(...);
         …..
    }
    catch(Mi_Exception_B  b)
    {
          Zona de reparación de las excepciones
          de tipo Mi_Exception_B
    }
    …..
```

1 lanza una excepción de tipo **Mi_Exception_B**

2 Atrapa la excepción del tipo **Mi_Exception_B**

Excepciones: Interrogantes...

- Cómo se lanzan las excepciones?
- Cómo se crean nuevas excepciones?
- Qué hay dentro una excepción?
- Cómo se usan las excepciones propias de Java?

Cómo se lanzan las excepciones?

La cláusula **throw**

Para provocar el lanzamiento de una excepción se utiliza la cláusula **throw**

```
metodo(...)throws Mi_Exception()
{
   ...
   throw new Mi_Exception();
}
```

Se crea un objeto de tipo Exception y se lo lanza

El método que cree y lance una excepción (que no sea capturada dentro del mismo método), debe declarar en su cabecera la cláusula throws con la correspondiente excepción.

Cómo se lanzan las excepciones?

La cláusula **`throw`**

Es posible lanzar excepciones comprobadas que sean extensiones del tipo de excepción declarado en la cláusula **`throws`**, ya que una clase se puede usar polimórficamente allí donde se espera a su superclase.

Cómo se lanzan las excepciones?

Las cláusulas **throw y throws**

Sólo se puede lanzar un tipo de excepción comprobada que haya sido declarado en la cláusula **throws** .

Si un método no tiene cláusula **throws**, no significa que se pueda lanzar cualquier excepción, sino que no se pueden lanzar excepciones comprobadas.

RuntimeException y **Error** son las únicas excepciones que no hace falta incluir en nuestras cláusulas **throws**. Son ubicuas, y cualquier método puede lanzarlas.

Los constructores pueden declarar y lanzar excepciones comprobadas.

Cómo se lanzan las excepciones?

Ejemplo:

Supongamos tener una colección de objetos de tipo Alumno, y un método propio de la colección que permita buscar un alumno (por legajo) y cambiar alguno de sus datos.

Qué sucede si el legajo buscado no pertenece a ningún alumno de la colección?

Ejemplo

```
public void actualizar(int legajo, Alumno nuevo)
                                    throws NoExisteException
{
   Alumno buscado = this.buscar(legajo);

   if(buscado == null)
      throw new NoExisteException();
   else
      ...
}
```

Será una responsabilidad de método invocante buscar una solución al lanzamiento de esta excepción. No puede ser resuelto dentro del mismo ámbito de la función

Ejemplo

```
...
   try
   {  ...
      coleccionAlumnos.actualizar(legajo,alumno);
      ...
   }
   catch(NoExisteException e)
   {
      // Zona de solución de la excepción
   }
}
```

Flujo de control

new

Las excepciones son objetos, por lo tanto deben crearse antes de lanzarse.

transferencia de control

Si se lanza la excepción **no se regresa** al flujo normal de programa. Una vez que se produce una excepción, las acciones que hubiera detrás del punto donde la excepción se produjo no tienen lugar. La siguiente acción que ocurrirá estará o en un bloque **`finally`** o en un bloque **`catch`** que capture la excepción

3 posibilidades cuando un método lanza una excepción

Si se invoca a un método que tiene una excepción comprobada en su cláusula throws, existen tres opciones:

1. Capturar la excepción y gestionarla.

```
catch(Mi_Exception e)
```

2. Capturar la excepción y transformarla en una de nuestras excepciones lanzando una excepción de un tipo declarado en nuestra propia cláusula throws.

```
catch(Mi_Exception e)
{   ...
    throw new Otra_Exception();
}
```

3. Declarar la excepción en nuestra cláusula throws y hacer que la excepción pase por nuestro método. Propagar sin hacer nada.

```
metodo(...) throws Mi_Exception
```

Cláusula `throws` y redefinición de métodos

No se permite que los métodos de redefinición o de implementación declaren más excepciones comprobadas en la cláusula **`throws`** que las que declara el método heredado.

Se pueden lanzar subtipos de las excepciones declaradas ya que podrán ser capturadas en el bloque **`catch`** correspondiente a su supertipo.

Si una declaración de un método se obtiene en forma múltiple, la cláusula **`throws`** de ese método debe satisfacer todas las cláusulas **`throws`** adquiridas por herencia o implementación.

Dos formas de usar **`finally`** correctamente

La situación general es que tenemos dos acciones, por ejemplo **`pre`** y **`post`**, de forma que si ocurre **`pre`**, debe ocurrir **`post`**, independientemente de las acciones que ocurran entre **`pre`** y **`post`**.

Dos formas de usar `finally` correctamente

```
pre();
try
{
   //otras acciones
}
finally
{
  post();
}
```

- Si pre tiene éxito entraremos en el bloque try e independientemente de lo que ocurra tenemos garantizado que post se ejecutará.
- Por otra parte, si pre falla por alguna razón y lanza una excepción, entonces post no se ejecutará (Si está try)
- Es importante que pre aparezca fuera del bloque try.

Dos formas de usar **`finally`** correctamente

```
Object val = null;
try
{
  val = pre();
  //otras acciones

}
finally
{
  if(val != null)
     post();
}
```

La segunda forma:

pre devuelve un valor que se puede utilizar para determinar si se completó con éxito o no. Sólo si pre se completó con éxito se invoca a post en la cláusula **`finally`**

finally

```
...
try
{
   ...
   return
}
finally
{
   ...
   return
}
```

- No hay forma de salir de un bloque try sin ejecutar su cláusula finally.
- Qué pasa si finally tiene su propio return?
  - Se respeta este último, ingnorando la sentencia return del bloque try, independientemente si dentro de try se lanzó alguna excepción.

Cómo crear nuevas excepciones?

```
public class Mi_Excepcion extends
Exception
{
    private Tipo dato;

    public Mi_Excepcion(Tipo dato)
    {
        super("cartel para el progamador");

        this.dato = dato
    }

    public Tipo getDato()
    {
        return this.dato;
    }
}
```

Información que se usará en la zona de solución

Invoca al constructor de la superclase con una descripción en forma de cadena de texto de lo que ha sucedido.

Almacena una descripción de la causa del error comprensible para las personas y los datos que crearon el error

Cómo crear nuevas excepciones?

```
public class Mi_Exception
          extends Exception
{
    private Tipo dato;

    public Mi_Excepcion(Tipo dato)
    {
       super("cartel para el progamador");

       this.dato = dato
    }

    public Tipo getDato()
    {
       return this.dato;
    }
}
```

```
try
{   …
  Tipo dato = new Tipo(...);
  throw new Mi_Exception(dato);
}
catch(Mi_Exception e)
{
    Tipo dato = e.getDato();
    …
}
// ojo con esto
```

Al crearse la excepción, se inicializa con los datos necesarios para que luego, en la zona de recuperación, se pueda determinar una solución. **VER PROPAGACIÓN**

Ejemplo

```
public void actualizar(int legajo, Alumno nuevo)
                throws NoExisteException
{
  Alumno buscado = this.buscar(legajo);

  if(buscado == null)
   throw new NoExisteException(legajo);
  else
    ...
}
```

Ejemplo

```
try
 {...
  coleccionAlumnos.actualizar(legajo,alumno);
  ...
 }
 catch(NoExisteException e)
 {
  int legajo = e.getDato();
  //solución planteada usando el legajo
 }
}
```

Pregunta

Qué diferencia existe entre estas tres palabras:

1)final

2)finalize

3)finally