IPOO

- Docentes
  - Profesores Adjuntos:
    - Carlos Di Cicco.
  - JTPs:
    - Federico Naso (Junín) .
      Nelson Di Grazia (Pergamino) .

Excepciones

Una excepción es un evento o problema que ocurre durante la ejecución de un programa y que interrumpe el flujo normal de ejecución de instrucciones.

Una excepción, interrumpe el procesamiento normal porque no cuenta con la información necesaria para resolver el problema, en el contexto en que sucedió.

Todo lo que se puede hacer es abandonar dicho contexto y pasar el problema a un contexto de más alto nivel.

Excepciones

Java usa excepciones para proveer manejo de errores a sus programas. Ej.: acceso a posiciones inválidas de un arreglo, falta de memoria en el sistema, ejecutar un query en una tabla inexistente de una bd, etc.

Cuando se dispara una excepción, ocurre lo siguiente:

1) se crea un objeto excepción en la heap, con el operador new, como cualquier objeto Java,
2) luego, se interrumpe la ejecución y el objeto excepción es expulsado del contexto actual. En este punto, comienza a funcionar el mecanismo de manejo de errores: buscar un

lugar apropiado donde continuar la ejecución del programa; el lugar apropiado es el manejador de excepciones, cuya función es recuperar el problema.

Excepciones. Clasificación

- Verificadas en Compilación (checked exception):

  son errores que el compilador verifica y que pueden recuperarse. Java requiere que los métodos que disparán excepciones, las capturen y manejen el error o, especifiquen todas las excepciones checked que pueden producirse dentro de su alcance.

- No-Verificadas en Compilación (unchecked exception):

  representan errores de programación difíciles de preveer. JAVA no obliga que éstas excepciones sean especificadas, ni capturadas para su manejo.

Ejemplo

```
import java.io.*;
public class InputFile {
  private FileReader in;
                    throws FileNotFoundException
  public InputFile (String filename)  {
      in=new FileReader(filename);
  }
                    throws IOException
  public String getWord()  {
      int c;
      StringBuffer buf=new StringBuffer();
      do {
          c=in.read();
          if (Character.isWhitespace((char)c))
              return buf.toString();
          else
              buf.append((char)c)
      } while (c!=-1)
      return buf.toString();
  }
}
```

Si compilamos la clase InputFile, el compilador nos da los siguientes dos errores:

InputFile.java: 11: Warning: Exception **java.io.FileNotFoundException** must be caught, or it must be declared in throws clause of this method.

```
in=new FileReader(filename);
```

InputFile.java: 19: Warning: Exception **java.io.IOException** must be caught, or it must be declared in throws clause of this method.

```
c=in.read();
```

El compilador detecta que tanto el constructor de la clase InputFile como el método getWord, no especifican ni capturan las excepciones que se generán dentro de su alcance, por lo tanto rechaza la compilación.

Ejemplo

in=new FileReader(filename);	c=in.read();
El nombre pasado como parámetro al constructor de la clase FileReader no existe en el file system, por tanto el constructor disparará la excepción: **java.io.FileNotFoundException**.	El método getWord() de la clase InputFile, lee del objeto FileReader abierto en el constructor de la clase, usando el método read(). Este método dispara la excepción: **java.io.IOException** si por algún motivo no se puede leer.

Al disparar estas excepciones, el constructor y el método read() de la clase FileReader permiten que los métodos que los invocan capturen dicho error y lo recuperen de una manera apropiada.

La clase InputFile, ignora que el constructor y método read() de la clase FileReader disparan excepciones. Sin embargo, el compilador JAVA obliga a que toda excepción checked sea capturada ó especificada. Por lo tanto, la clase InputFile no compila.

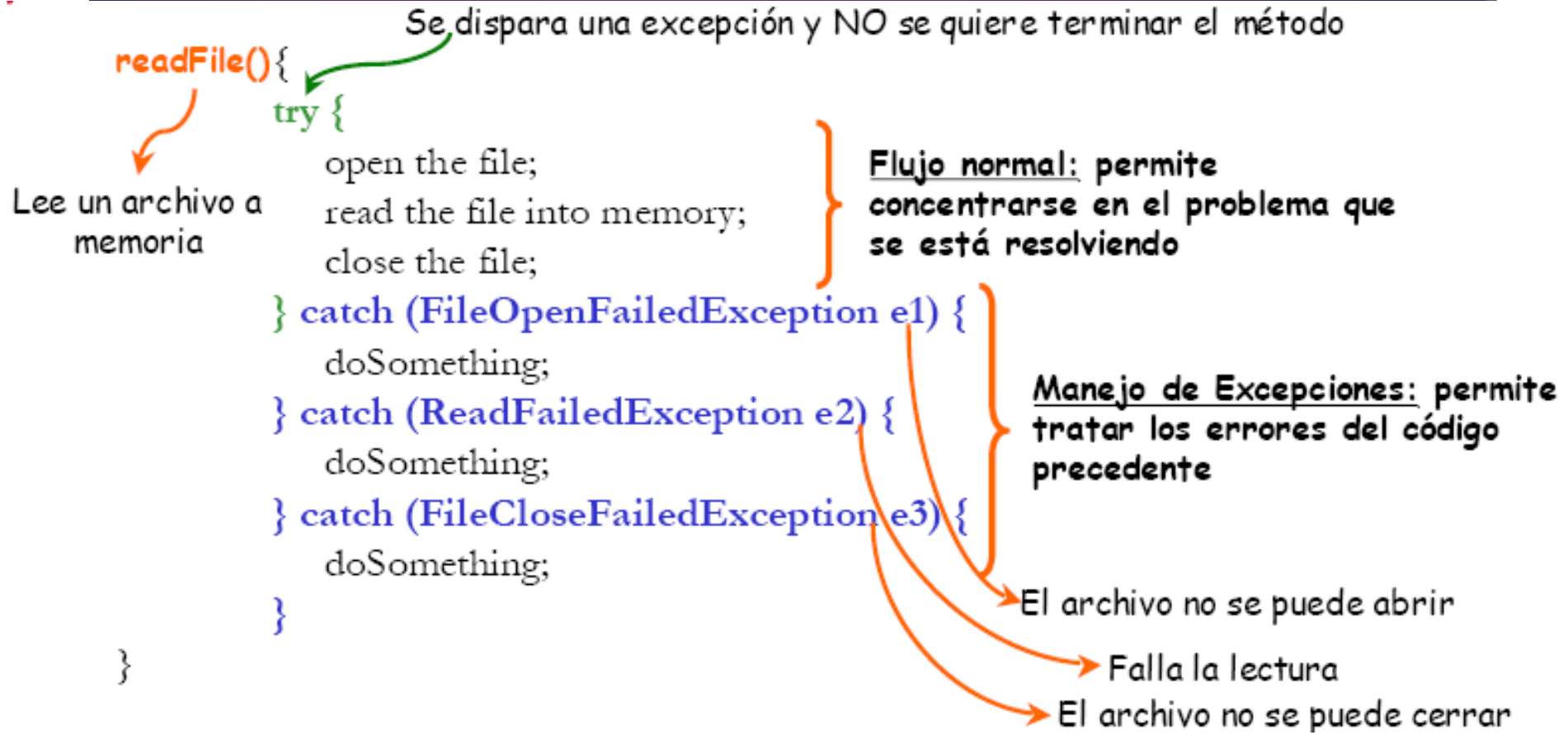
Ejemplo

En este punto tenemos dos opciones:

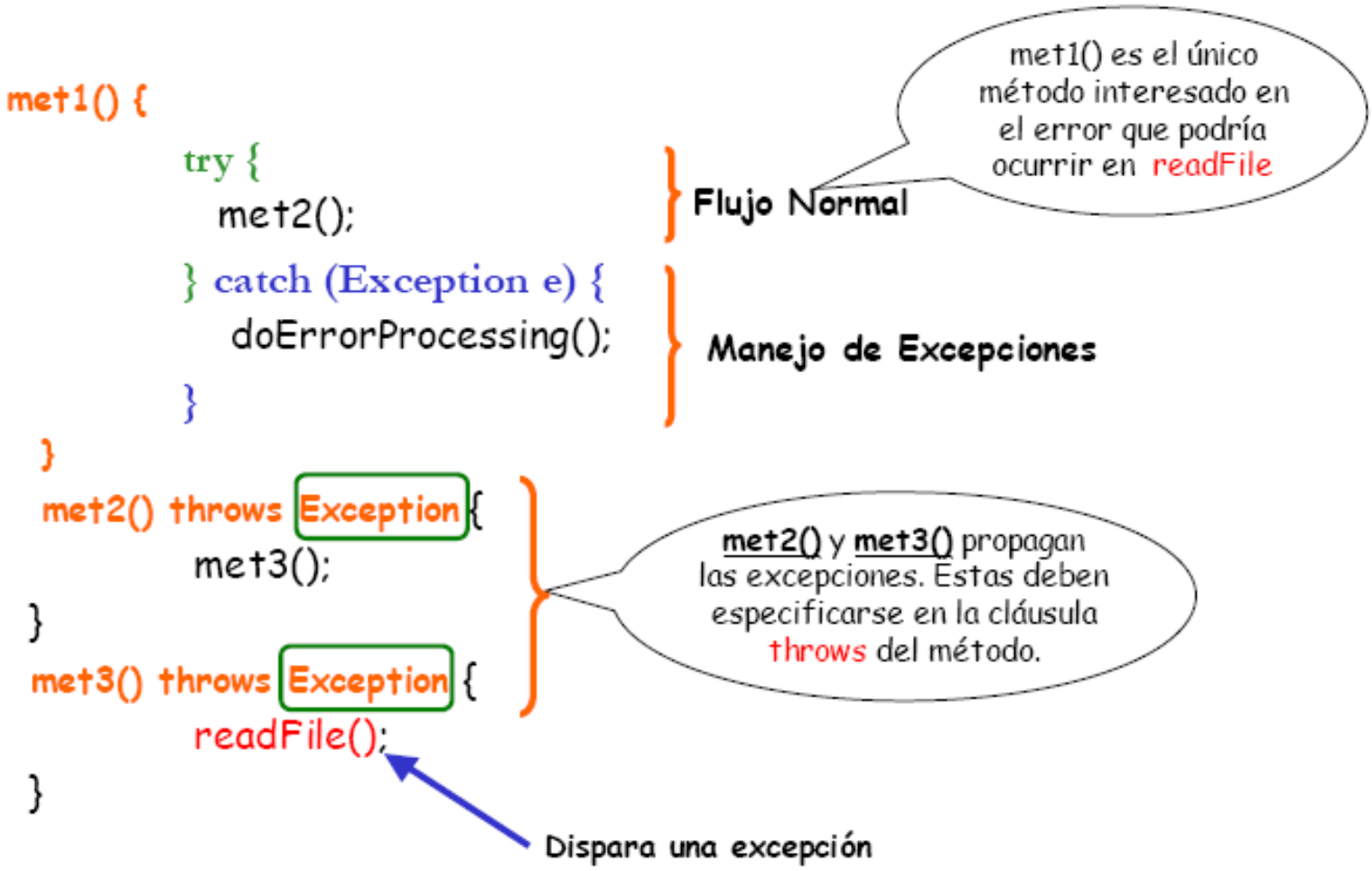
1) Arreglar al constructor de la clase InputFile y al método getWord() para que capturen y recuperen el error, ó

2) Pasar por alto los errores y darle la oportunidad a los métodos que invoquen al constructor y a los métodos de InputFile a que recuperen los errores (usando la claúsula throws en el encabezamiento).

Capturar Excepciones



Propagar excepciones



Jerarquía de Clases de Excepciones

Throwable es la clase base de todas las excepciones, describe todo lo que puede dispararse como una excepción.

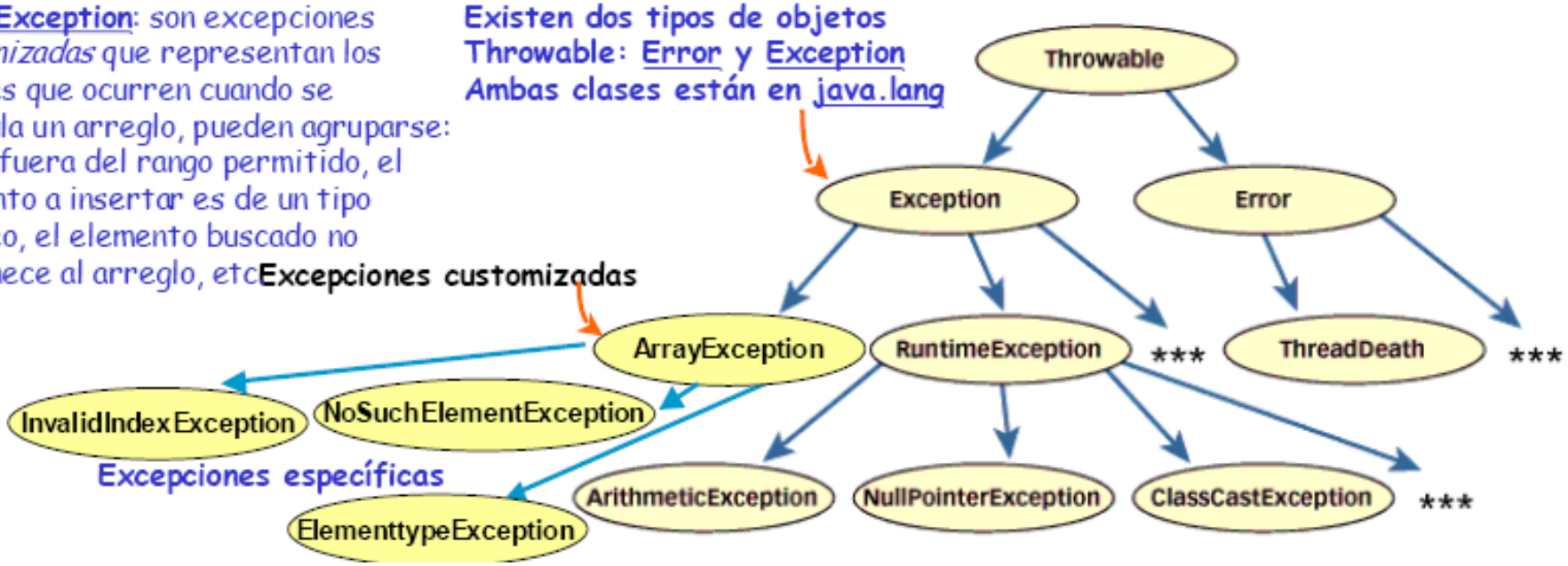
Exception es el tipo base de todos los objetos que pueden dispararse desde cualquier método de la API de Java o desde nuestros propios métodos, cuando ocurren condiciones anormales, que frecuentemente pueden ser manejadas por un código específico y pueden ser recuperadas.

Error representa problemas serios, relacionados con la computadora, la memoria o el procesador. Los errores son disparados por la JVM y los programadores no pueden hacer nada.

Jerarquía de Clases de Excepciones

ArrayException: son excepciones *customizadas* que representan los errores que ocurren cuando se manipula un arreglo, pueden agruparse: índice fuera del rango permitido, el elemento a insertar es de un tipo erróneo, el elemento buscado no pertenece al arreglo, etc

Existen dos tipos de objetos Throwable: Error y Exception
Ambas clases están en java.lang



RunTimeException

- Representan errores de programación, que no es posible anticiparlos, por ejemplo una referencia nula, el acceso a una posición fuera del rango del arreglo, división por cero, etc.

- Este tipo de excepciones son subclase de RunTimeException. Son excepciones no-verificadas o unchecked exceptions, el compilador no fuerza a especificarlas, no puede detectarlas.

- Son disparadas automáticamente por la JVM

El bloque try

```
try {
        sentencias JAVA
}
```

Las sentencias JAVA que pueden disparar excepciones deben estar encerradas dentro de un bloque try.

```
PrintWriter out=null;
try{
    out=new PrintWriter(new FileWriter("OutFile.txt"));

   for (int i=0; i<v.size; i++)
        out.println("Valor en: "+ i +" = "+v.elementAt(i));
}
```

Dispara un IOException si no puede abrirse el archivo

Dispara un ArrayIndexOutOfBoundsExcep si el índice es muy chico (número negativo) ó muy grande

RunTimeException

catch

-Son manejadores de excepciones

-La forma de asociar manejadores de excepciones con un bloque try es proveyendo uno ó más bloques catch inmediatemente después del bloque try.

catch

```
try{
    .....
} catch (SQLException e) {
    System.err.println("Excepción capturada..." + e.getMessage());
} catch (IOException e) {
    System.err.println("Excepción capturada..." + e.getMessage());
}
```

catch

-Si se dispara una excepción, el mecanismo de manejo de excepciones comienza a buscar el primer manejador con un argumento que coincida con el tipo de excepción disparada. La coincidencia entre el tipo de la excepción y la de su manejador, no tiene que ser exacta. El tipo del manejador, puede ser cualquier super-clase de la excepción disparada.

-Luego, se ejecuta el bloque catch y la excepción se considera manejada. Solamente se ejecuta el bloque catch que coincide con la excepción disparada.

-Si adentro del bloque try , la invocación a diferentes métodos, dispara el mismo tipo de excepción, solamente necesitamos un único manejador de excepciones.

catch

```
catch (FileNotFoundException e) {
        .....
}
```

Manejador de excepciones específico

Otra forma, es capturar un error basado en su grupo ó tipo general, especificando alguna de las superclases de excepciones.

```
catch (IOException e) {
        .....
}
```

Se puede averiguar la excepción específica usando el parámetro e

Se puede establecer un manejador muy general que capture cualquier tipo de excepciones. Deben ubicarse al final de la lista de manejadores.

```
catch (Exception e) {
        .....
}
```

Los manejadores de excepciones muy generales hacen el código propenso a errores pués capturan y manejan excepciones que no fueron previstas y por lo tanto no se manejan correctamente. No son útiles para recuperación de errores

catch

La clase Throwable, superclase de Exception, provee de un conjunto de métodos útiles para obtener información de la excepción disparada:

- String getMessage(): devuelve un mensaje detallado de la excepción
- String getLocalizedMessage(): idem getMessage(), pero adaptado a la región
- String toString(): devuelve una descripción corta del Throwable incluyendo el mensaje (si existe)

finally

El último paso para definir un manejador de excepciones es liberar recursos antes de que el control sea pasado a otra parte del programa. Esto se hace, escribiendo el código necesario para liberar recursos adentro del bloque finally.

El sistema de ejecución de Java, siempre ejecuta las sentencias del bloque finally independientemente de lo que sucedió en el bloque try .

Ejemplo

```
public void writeList() {
        PrintWriter out=null;
        try {
              out=new PrintWriter(new FileWriter("OutFile.txt"));
              for (int i=0; i<v.size; i++)
              out.println("Valor en: "+ i +" = "+v.elementAt(i));
        } catch (ArrayIndexOutOfBoundsException e) {
                   System.err.println("Excepción capturada..." + e.getMessage());
        } catch (IOException e) {
                  System.err.println("Excepción capturada..." + e.getMessage());
       } finally {
         if (out !=null){
              System.out.println("Cerrando PrintWriter");
              out.close();
         } else {
             System.ou.println("PrintWriter no fue abierto");
         }
      }
}
```

Especificación de Excepciones

Java provee una sintáxis (y fuerza a usarla) que permite informarle al programador que excepciones podría disparar, de manera tal que el programador puede manejarlas.

Especificación de excepciones.

- La Especificación de excepciones es parte de la declaración del método y aparece después de

la lista de argumentos. Usa la palabra clave throws seguida por una lista de tipos de excepciones.

Especificación de Excepciones

Si un método NO captura y maneja las excepciones checked disparadas dentro de su alcance, el compilador Java fuerza al método a especificarlas en su declaración.

En algunas situaciones es mejor que un método propague las excepciones, por ejemplo si se está implementando un paquete de clases, es posible que no se puedan preveer las necesidades de todos los usuarios del paquete.

En este caso, es mejor no capturar las excepciones y permitir a los métodos que usan las clases que manejen las excepciones que pueden dispararse.

Especificación de Excepciones

```
public void writeList() throws IOException, ArrayIndexOfBoundsException {
        PrintWriter out=null;
        out=new PrintWriter(new FileWriter("OutFile.txt"));

       for (int i=0; i<v.size; i++)
       out.println("Valor en: "+ i +" = "+v.elementAt(i));
        out.close();
}
```

No es necesario especificarla pués es una RunTimeException

¿Cómo disparar Excepciones ?

La palabra clave throw es usada por todos los métodos que disparán excepciones y requiere como único argumento un objeto throwable.

¿Cómo disparar Excepciones ?

El método pop() usa la claúsula **throws** para declarar que puede disparar un una **EmptyStackException**

```
public Object pop() throws EmptyStackException {
        Object obj;
        if (size ==0)
                        throw new EmptyStackException();
        obj = objectAt(size -1);
        setObjectAt(size –1, null);
        size--;
       return obj;
}
```

- Se crea un objeto en la *heap* que representa el error y la referencia la tiene la claúsula **throw**
- El objeto es EmptyStackexception es retornado por el método pop()

El método pop() chequea si hay algún elemento en la pila. Si está vacía, el método pop() instancia un objeto **EmptyStackException**

Restricciones en Excepciones

-Cuando se sobreescribe un método, se pueden disparar solamente las excepciones especificadas en la versión de la clase base del método. La utilidad de esta restricción es que el código que funciona correctamente para un objeto de la clase base, seguirá funcionadando para un objeto de la clase derivada (principio fundamental de la OO)

-La interface de especificación de excepciones de un método puede reducirse y sobreescribirse en la herencia, pero nunca ampliarse. Es exactamente opuesto a lo que ocurre en la herencia, con la interface de una clase.

Restricciones en Excepciones

Constructores

-Los constructores no se sobreescriben.

-Los constructores de una subclase pueden disparar excepciones diferentes a las excepciones disparadas por el constructor de la superclase.

-Hay que ser cuidadoso de dejar el objeto, que se intentá construir y no se puede, en un

estado seguro.

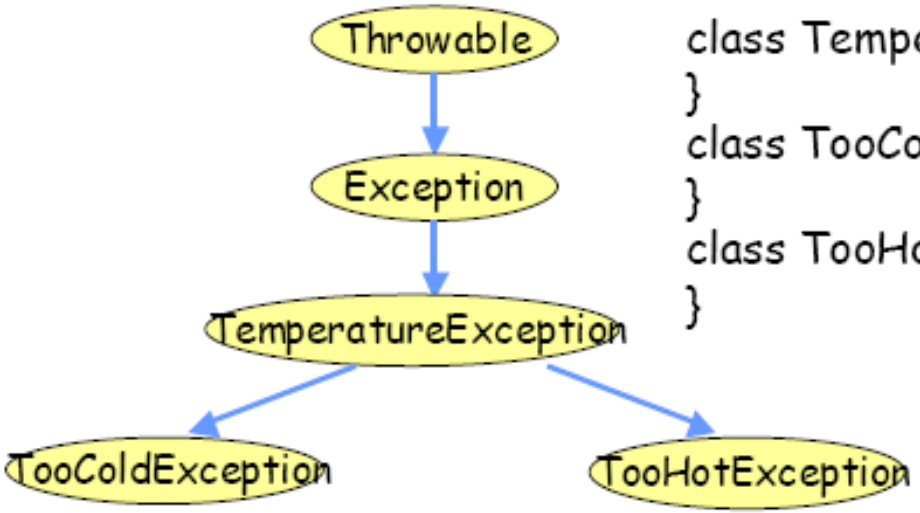
Crear Clases de Excepciones

Ejemplo: Café Virtual

Cuando se diseña un paquete de clases JAVA, éstas deben interactuar bien y sus interfaces deben ser fáciles de entender y de usar. Para ello, es bueno diseñar clases de excepciones.

Las condiciones excepcionales que pueden ocurrir cuando el cliente toma una taza de café son las siguientes: el café está muy frío o muy caliente.

Crear Clases de Excepciones




```
class TemperatureException extends Exception {
}
class TooColdException extends TemperatureException {
}
class TooHotException extends TemperatureException {
}
```

Convención de nombres: es una buena práctica agregar el string Exception a todos los nombres de clases que heredan directa ó indirectamente de la clase Exception.

Crear Clases de Excepciones

```
class VirtualPerson {
  private static final int tooCold = 65;
  private static final int tooHot = 85;

  public void drinkCoffee(CoffeeCup cup) throws
     TooColdException, TooHotException {

     int temperature = cup.getTemperature();

     if (temperature <= tooCold) {
        throw new TooColdException();
     }

     else if (temperature >= tooHot) {
        throw new TooHotException();
     }
  }
}
```

Se declaran las excepciones que puede disparar el método **drinkCoffee()**

Disparo de una Excepción

```
class CoffeeCup {
// 75 grados Celsius: es la temperatura ideal del café
   private int temperature = 75;

   public void setTemperature(int val){
      temperature = val;
    }

   public int getTemperature() {
      return temperature;
   }

}
```

Ejemplo Café Virtual

```
class VirtualCafe {
   public static void serveCustomer(VirtualPerson cust, CoffeeCup cup) {
   try {
         cust.drinkCoffee(cup);
         System.out.println("El Café está OK.");

      } catch (TooColdException e) {
         System.out.println("El Café está muy frío.");
      } catch (TooHotException e) {
         System.out.println("El Café está muy caliente.");
      }
   }

}
```

El método drinkCoffee puede disparar las excepciones: **TooHotException ó TooColdException**

```
catch (TemperatureException e) {
   System.out.println("El Café no está OK");
}
```

Ejemplo Café Virtual

- Se recomienda el uso de manejadores de excepciones especializados.
- Los manejadores genéricos (que agrupan muchos tipos de excepciones) no son útiles para recuperación de errores, ya que el manejador tiene que determinar que tipo de excepción ocurrió, para elegir la mejor estrategia para recuperar el error.
- Los manejadores genéricos pueden hacer que el código sea más propenso a errores, ya que se

capturan y manejan excepciones que pueden no haber sido previstas por el programador.

Incorporar información a las Excepciones

- Las excepciones además de transferir el control desde una parte del programa a otra permiten transferir información.
- Es posible agregar información a un objeto excepción acerca de la condición anormal que se produjo
- La claúsula catch permite obtener información interrogando directamente al objeto excepción.
- La clase Exception permite especificar mensajes de tipo String a un objeto excepción y, recuperarlos vía el método getMessage() (sobre el objeto excepción).
- Es posible agregar a un objeto Exception información de un tipo distinto que String. Para ello, es necesario agregar a la subclase de Exception datos y métodos de acceso a los mismos.

Ejemplo

```
class UnusualTasteException extends Exception {
    UnusualTasteException() {
    }
    UnusualTasteException(String msg) {
        super(msg);
    }
}
```

Dos constructores para UnusualTasteException

Un programa que dispara una excepción de tipo *UnusualTasteException*, puede hacerlo de las dos formas siguiente:

a) `throw new UnusualTasteException()`
b) `throw new UnusualTasteException("El Café parece Té")`

```
try {
        .....
    } catch (UnusualTasteException e) {
        String s = e.getMessage();
        if (s != null) {
            System.out.println(s);
    }
}
```

Se obtiene información del objeto excepción

Incorporar información a las excepciones

```
abstract class TemperatureException extends Exception {
private int temperature;

public TemperatureException(int temperature) {
        this.temperature = temperature;
    }
public int getTemperature() {
        return temperature;
    }
}
```

Datos y métodos de acceso a la información asociada a la excepción

```
class TooColdException extends TemperatureException {
    public TooColdException(int temperature) {
        super(temperature);
    }
}
```

```
class TooHotException extends TemperatureException {
    public TooHotException(int temperature) {
        super(temperature);
    }
}
```

Ejemplo Café Virtual

```
class VirtualPerson {

   private static final int tooCold = 65;
   private static final int tooHot = 85;

   public void drinkCoffee(CoffeeCup cup) throws
         TooColdException, TooHotException {

      int temperature = cup.getTemperature();
      if (temperature <= tooCold) {
         throw new TooColdException(temperature);
      }
      else if (temperature >= tooHot) {
         throw new TooHotException(temperature);
      }
      //...
   }
   //...
}
```

```
class VirtualCafe {
public static void serveCustomer(VirtualPerson cust
                                 CoffeeCup cup) {

 try {
       cust.drinkCoffee(cup);
     } catch (TooColdException e) {
       int temp = e.getTemperature();
       if (temp > 55 && temp <= 65) {
       } else if (temp > 0 && temp <= 55) {
       } else if (temp <= 0) {
         .....
       }
     } catch (TooHotException e) {
       int temp = e.getTemperature();
       if (temp >= 85 && temp < 100) {
       } else if (temp >= 100 && temp < 2000) {
       } else if (temp >= 2000) {
       }
     }
 }
}
```

Generics

- Los tipos genéricos (generics) permiten asignar parámetros a las clases, interfaces, métodos..., de forma que sólo admitan los tipos de objetos que uno necesita.

- No se pueden utilizar con tipos de datos primitivos, pero si con las clases que se corresponden con ellos. Por eso no puedes hacer un genérico tipo int, pero sí un Integer.

Generics

- Ejemplo de clase genérica

```
public class Box<T>{
   private T t;
   public void set (T t){
   this.t = t;
   }
   public T get( ){
   return t;
   }
}
```

Generics

- Para instanciar esta clase, use la palabra clave new, como es usual, pero ponga el tipo de dato (<Integer> en este ejemplo) entre el nombre de la clase y el paréntesis:

- Box<Integer> IntegerBox = new Box<Integer>();

Generics

- Una vez que se inicializa integerBox, se invoca libremente al método get sin necesidad de hacer un casteo (cast)

```
public static void main(String[] args) {
Box<Integer> integerBox = new Box<Integer>();
Integer someInteger = integerBox.get(); // no cast!
System.out.println(someInteger);    }
```

Generics

Tener en cuenta que un tipo genérico puede tener múltiples parámetros de tipo, pero cada parámetro debe ser único en su clase o interfaz declarante.

Una declaración de Box<T,T>, por ejemplo, generaría un error en la segunda ocurrencia de T, pero Box<T,U>, sin embargo, sería permitido.

Generics. Convenciones de Nombramiento de Parámetros de Tipo

Por convención, los nombres parámetros de tipo son una sola letra mayúscula. Sin esta convención, sería difícil determinar la diferencia entre una variable de tipo y el nombre de una clase o interfaz ordinaria.

Los parámetros de tipo más comúnmente usados son:

- E - Elemento de Colecciones Java
- K - Clave
- N - Número
- T - Tipo
- V - Valor
- S,U,V etc. - tipos 2°, 3°, 4°