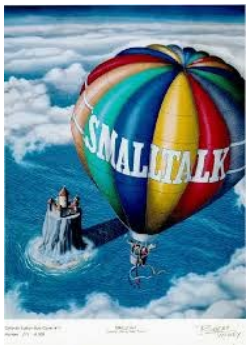
Departamento de Programación
Facultad de Informática
Universidad Nacional del Comahue

Programación Concurrente



Excepciones

Excepciones

- Un programa **correcto** es aquel que actúa de acuerdo a su **especificación**.

Algún evento inesperado o no deseado tarde o temprano ocurrirá cuando un sistema este ejecutándose”

- Un evento anormal no necesariamente es catastrófico y con frecuencia **puede repararse** de modo tal que la ejecución continúe.
- El software que previene este tipo de circunstancias se dice "tolerante a las fallas".

¿Cómo avisar que hubo un error?

- En algunos casos se puede devolver un valor especial
  - Ejemplo: El método *indexOf(cad)* de String devuelve -1 cuando no encuentra cad en el String llamador
- En otros casos no es posible
  - Ejemplo: dividir(Entero) → Entero
    No hay ningún valor entero para indicar que hubo un error si el segundo parámetro es 0

Excepciones

- Se puede **reparar** la falla, **capturando** la excepción y alcanzando un estado que permita continuar la ejecución.

- A veces, el manejo de la excepción se reduce a mostrar un mensaje, porque la situación no es recuperable.

- En ese caso la operación falla y probablemente el programa se aborta.

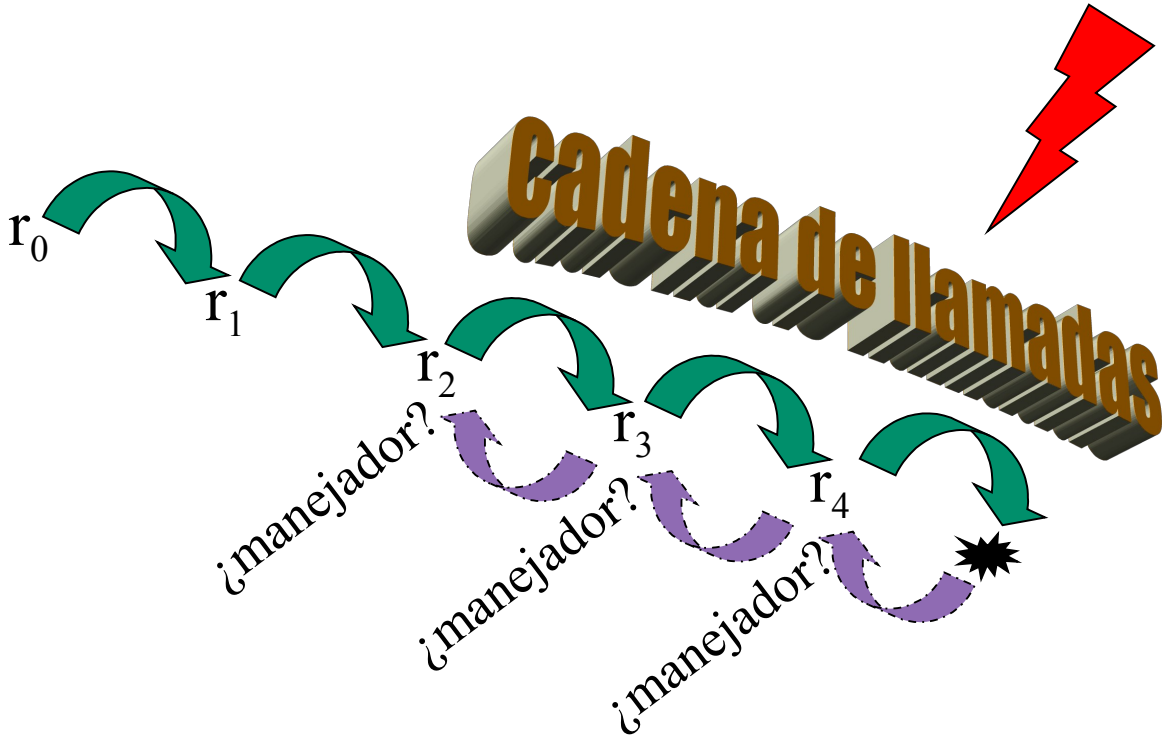
Excepciones

- Una excepción es una situación anormal o poco frecuente que requiere ser **capturada** y **manejada** adecuadamente.
- Las excepciones pueden ser **predefinidas por el lenguaje** o **definidas por el programador**.
- Las excepciones predefinidas son más generales y son **capturadas implícitamente** por alguna operación predefinida.

Manejo de excepciones

Después que se levanta la excepción el sistema debe buscar el manejador de la excepción.

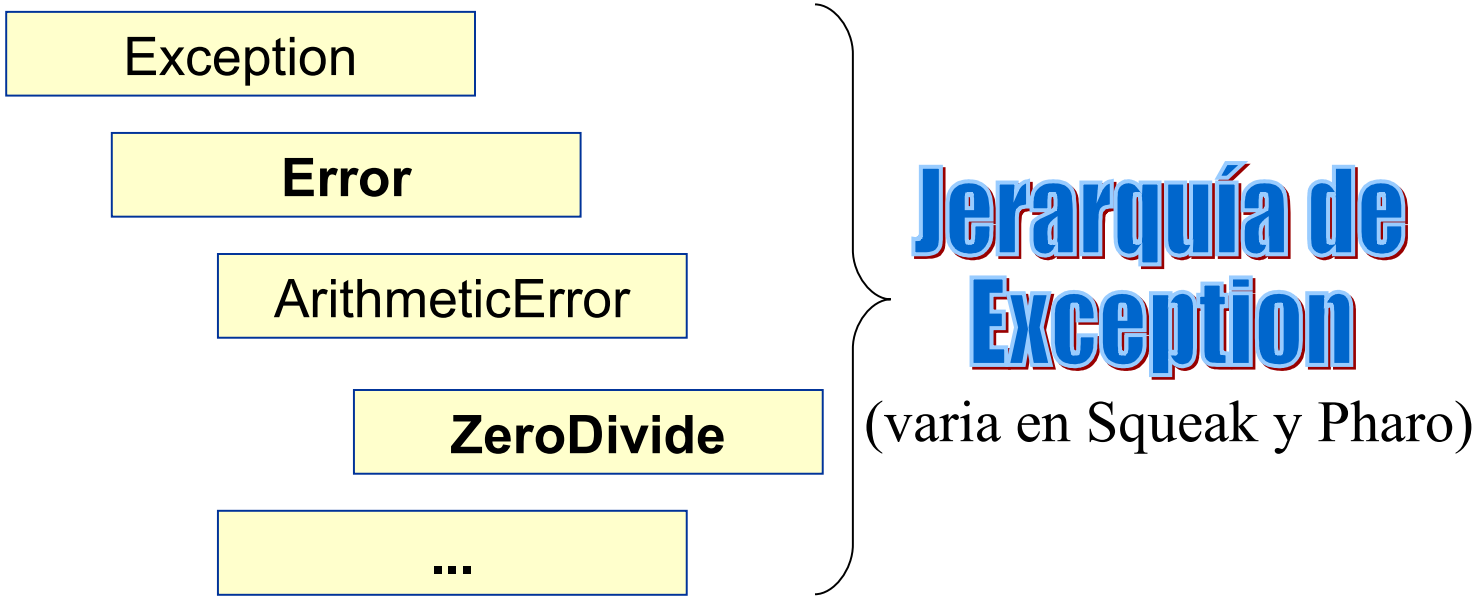
¿Cúales son los métodos capaces de manejar la excepción?



Organizar programas

- Organizar un programa en secciones para el caso normal y para el caso excepcional
  - Ejemplos: división por cero, entrada de datos de tipo incorrecto, etc
- Implementar los programas incrementalmente
  - Codificar y probar el código para la operación normal primero
  - Después agregar el código para el caso excepcional
- Tener en cuenta: las excepciones simplifican el desarrollo, prueba y mantenimento, pero no se debe abusar de ellas.

Excepciones en Smalltalk



SM, Invocación de Excepciones

- Cada clase "exception" define o hereda un mensaje de **acción por defecto** (defaultAction)
- El **defaultAction** es invocado cuando ocurre la excepción.
- Esto pasa a menos que esté definido un **manejador** (handler) para dicha excepción.

Manejador de Excepciones

- Para la mayoría de las excepciones la acción por defecto es *mostrar un notificador*.
- En un desarrollo es a veces necesario determinar la causal del error y repararla.
- Para una aplicación *no es apropiado* un notificador.
- La excepción necesita ser manejada por la aplicación propiamente dicha.
- Para manejar las excepciones en una aplicación se debe definir un *manejador de excepciones*.

SM:Manejador de Excepciones

Bloque protegido
(que se desea proteger de posibles errores)

[bloque de código]

on: ***TipoExcepcion***

do: ***bloque de código***

Con la excepción como argumento

Bloque manejador
(que se desea hacer para tratar la excepción)

Una división por cero retornaría:

'está mal, no se puede'

```
nuevoFactor := [x / y]
        on: Error
        do: [:ex| 'esta mal, no se puede' ].
```

Squeak

- Las excepciones y errores son representados como instancias de clases en la **Jerarquía de Excepciones**
- La jerarquía comienza con **Object\Exception**.
- Dos de las subclases más importantes son **Error,** y **Notification**.
- Las subclases de las clases de excepciones definen otras excepciones más específicas que pueden ser obtenidas a partir de la aplicación.
- Cada uno puede definir **subclases de Excepciones** para errores y excepciones específicas.

¿Qué puede hacer el manejador?

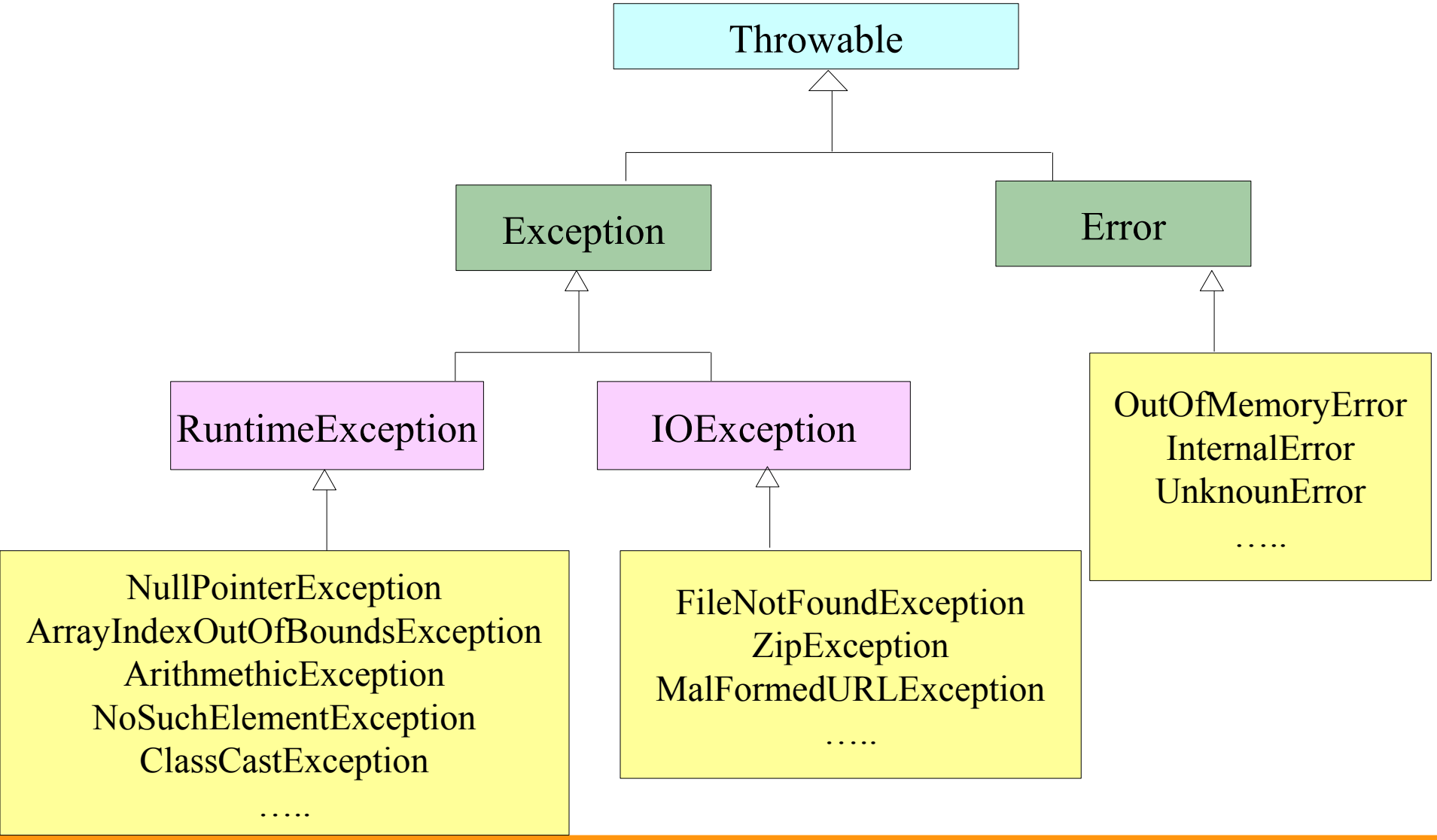
- Mensajes que pueden ser enviados al argumento del bloque manejador, o sea a la instancia de excepción creada:
  - ***resume*** o ***resume:*** intenta continuar procesando el bloque protegido, inmediatamente después de levantada la excepción.
  - ***return*** o ***return:*** termina el procesamiento del bloque que levantó la excepción, sin manejarla.
  - ***retry*** reevalúa el bloque protegido con los nuevos valores.
  - ***retryUsing:*** evalúa un Nuevo bloque en lugar del bloque protegido.
  - ***pass*** sale del manejador y pasa al siguiente manejador, hacia fuera en la cadena de llamadas

Manejadores anidados

Ejemplo

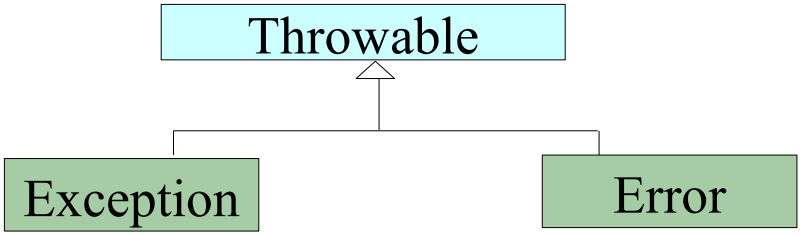
```
[ bloque protegido 1
        [ bloque protegido 2
                [ bloque protegido 3
                        [ bloque protegido 4  ]
                                on: ColorError
                                do: [ bloque manejador 4 ] ]
                        on: Warning
                        do: [ bloque manejador 3 ] ]
                on: Error
                do: [bloque manejador 2] ]
        on: ZeroDivide
        do: [ bloque manejador 1 ]
```

Java, clases para manejo de excepciones



Clases de las Excepciones

- Throwable: Clase base que representa todo lo que se puede lanzar de Java



  - Contiene una instantánea del estado de la pila en el momento en el que se creó el objeto “stack trace” o “call chain”
  - Almacena un mensaje (tipo String) para detallar el error
- Error: indica problemas graves que una aplicación no debería intentar solucionar Ej. memoria agotada
- Exception: Situaciones que una aplicación debería tratar en forma razonable.

Excepciones detectadas implícitamente

- Ejemplos típicos de excepciones detectadas implícitamente son:
  - **ArithmeticException** División por 0, señalizado por la operación /
  - **ArrayIndexOutOfBoundsException** El acceso fuera de rango dentro de un arreglo, señalizado por la operación de subindización
  - **NullPointerException** Se intenta acceder a un servicio de una variable de tipo clase pero esta no está asociada a un objeto.

Manejadores (SM y Java)

- En general, cuando se captura la excepción aparece un mensaje de error y el programa termina anormalmente (aborta)
- La idea es que **el programador establezca un manejador** que especifique las acciones a realizar cuando se captura una excepción.
- La acción puede ser algo tan simple como mostrar un mensaje de error diferente al predefinido o puede de alguna manera 'salvar' la situación anormal para reparar la excepción.

Java: Terminología

- Lanzar o disparar una excepción (**throwing**)
  - Java por sí mismo o nuestro código señala cuando algo inusual pasa
- Manejar o capturar una excepción **(handling/catching**)
  - Se responde a una excepción ejecutando una parte del programa escrita específicamente para esa excepción
- El caso normal es manejado en un bloque **try**
- El caso excepcional es manejado en un bloque **catch**
- El bloque catch recibe un parámetro de tipo Exception (generalmente llamado e)
- Si se dispara una excepción, la ejecución del bloque **try** se interrumpe y el control pasa al bloque **catch** cercano al bloque try

Dupla **try-catch** con **throw**

```
try
{
    <código a tratar>
    obj.metodoAux(...)
    <más código>
}
catch(Exception e)
{
    <código de manejo de la excepción>
}

<posiblemente más código>
```

```
...
if(condición de prueba)
        throw new Exception
        ("Mensaje de error");
...
```

Excepciones múltiples y bloques **`catch`** en un Método

- Un método puede lanzar más de una excepción
- Los bloques **catch** immediatamente después del bloque try son analizados en secuencia para identificar el tipo de excepción
- El primer bloque catch que maneja ese tipo de excepción es el único que se ejecuta
- Se deben colocar los bloques catch en orden de especifidad: los más específicos primero

```
catch (ExcepcionDividePorCero e){
  // que hace si ocurre excepción divide por cero
}
 catch (Exception e){
 // aquí lo que hace si ocurre otra excepción
 }
```

La terna **try-catch-finally**

Organización básica del código

```
try{
    <más código>
}
catch(Exception e){
    <código de manejo de la excepción>
}
finally(Exception e){
<posiblemente más código>
}
```

SIEMPRE SE EJECUTARA **(sin importar si se produjo la excepción o no);**

Posibilidades de `try-catch-finally`

- El bloque try se ejecuta hasta el final sólo si ninguna excepción es lanzada.
  - El bloque finally se ejecuta después del bloque try.

    (bloque1- bloque3)

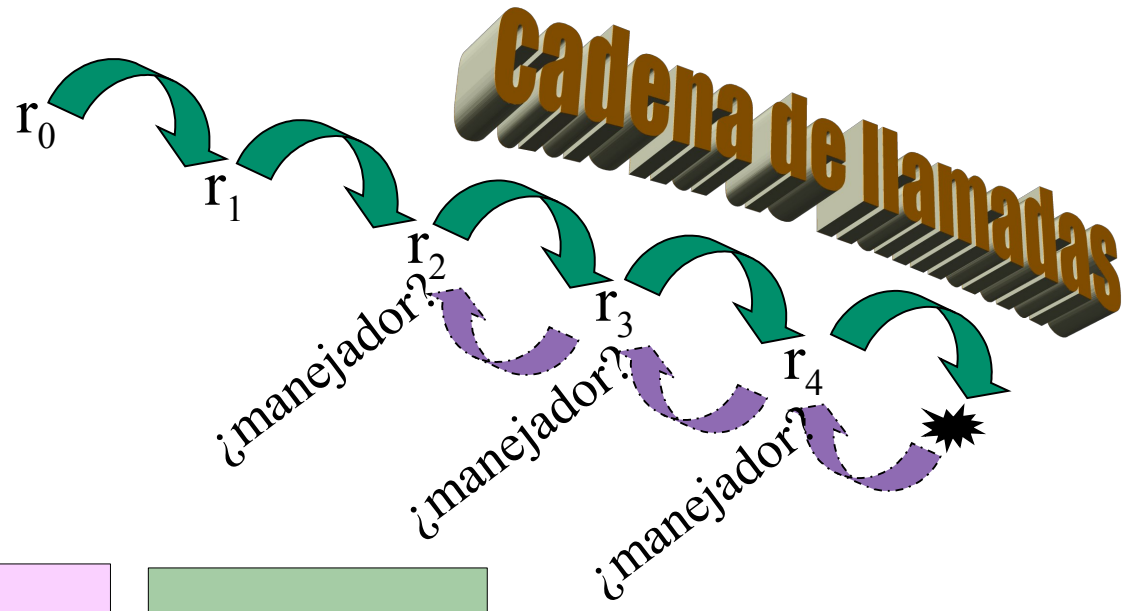
```
try {bloque1}
catch (tipoEx) {
        bloque2}
finally {bloque3}
```

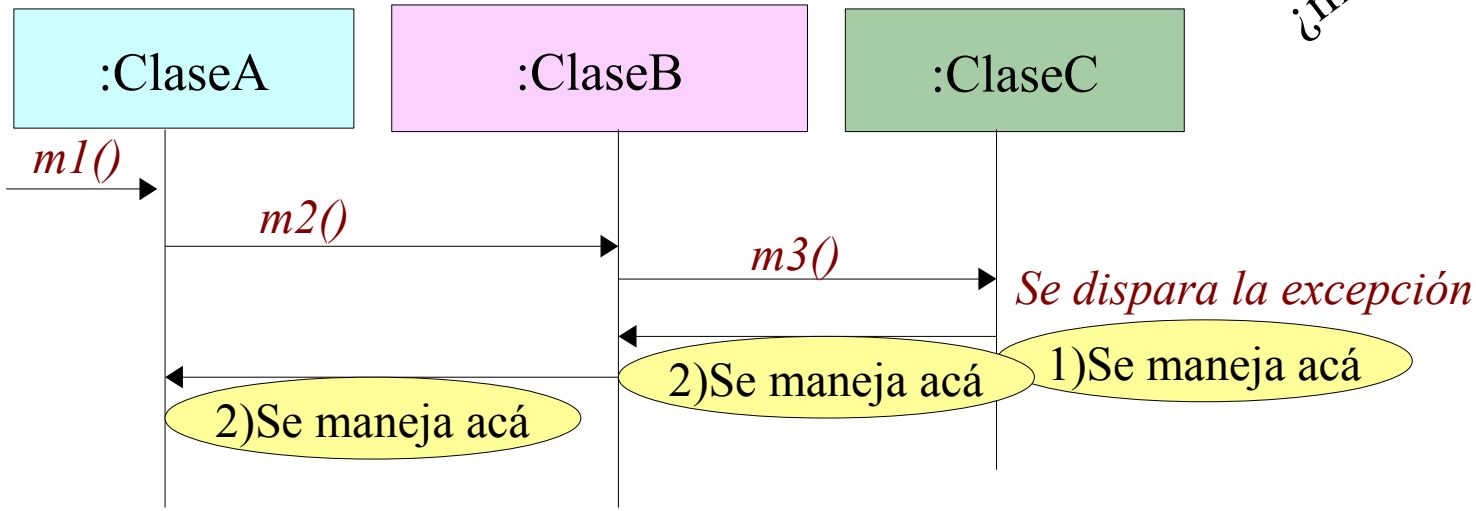
- Una excepción es lanzada en el bloque try y atrapada en el macheo del bloque catch.
  - El bloque finally se ejecuta después del bloque catch.

    (bloque1- bloque2 - bloque3)
- Una excepción es lanzada en el bloque try y no existe match en el bloque catch.
  - El bloque finally se ejecuta antes de que el método termine.
  - El código que está después del bloque catch pero no en el bloque finally no sería ejecutado en esta situación.

    (bloque1- bloque2 - bloque3)

Manejo de excepciones

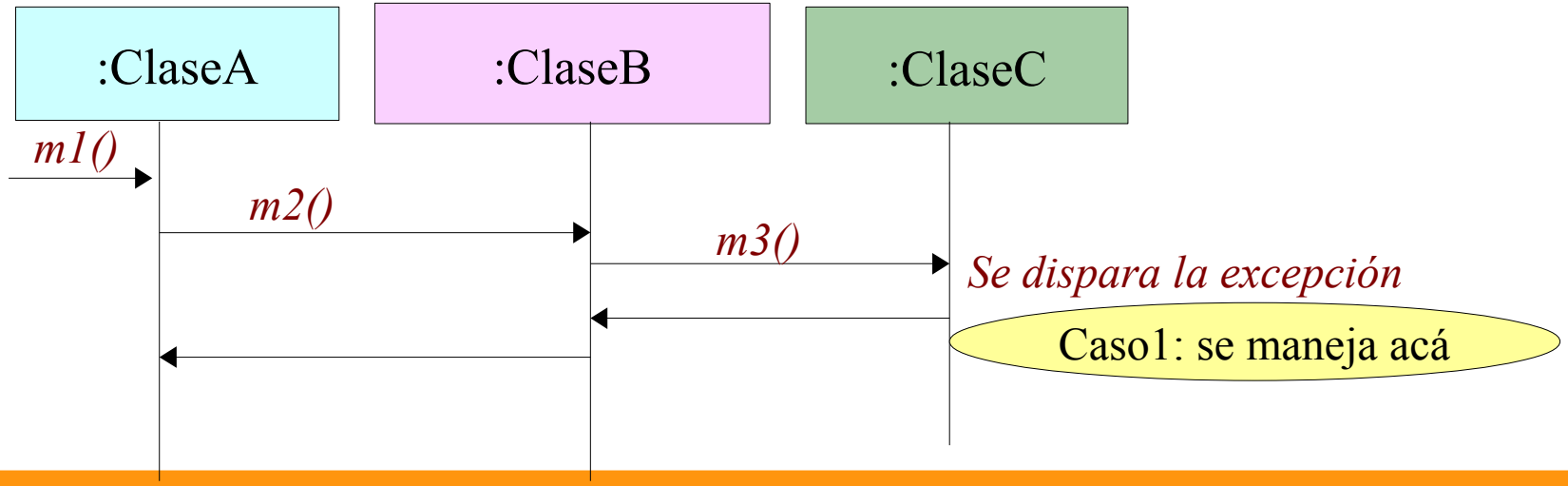




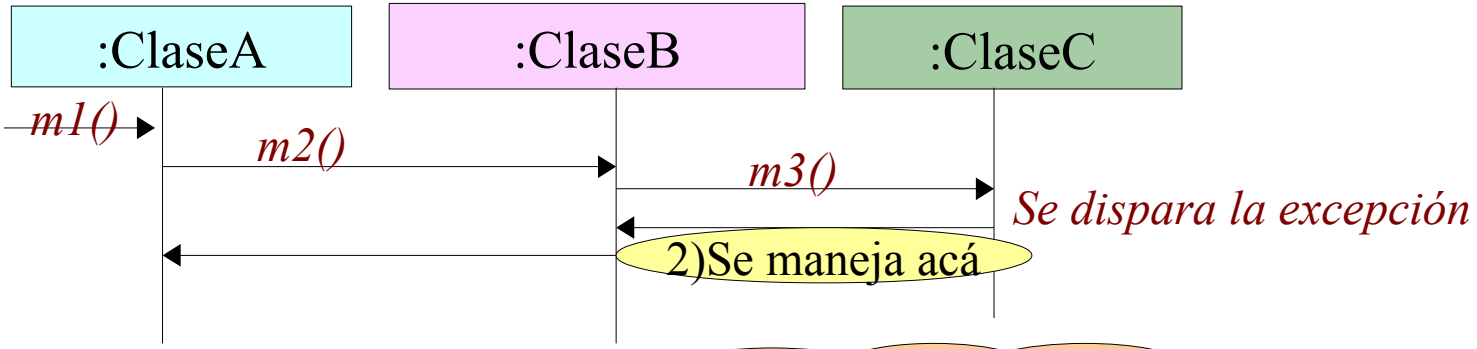
Manejo de excepciones

CASO 1

```
public static void m1(....){
   // fragmento libre de excepciones
   try {
   // codigo que puede lanzar una excepción tipo IOException
   ….
   }
   catch (IOException e){
   // tratamiento en caso que se produzca la excepción
   }
}
```



Propagación de excepciones



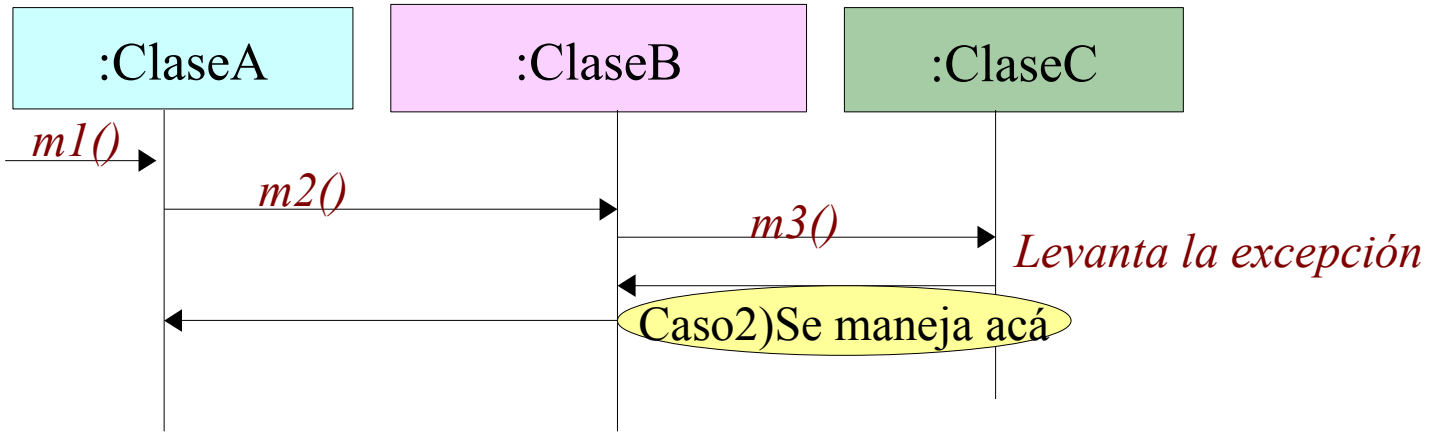
La excepción se propaga

Tipo de excepción

```
public ClaseC{

   public int m3(....) throws ErrorA {
      if (….)
         throw new ErrorA(1); //ErrorA es subclase de Exception
      else
         return ...
   }
}
```

Manejo de Exepciones (cont)




```
public class ClaseB{
  public static void m2(....){
    try {
      ClaseC.m3(...); // si disparó la execpción
      ….}
    catch (ErrorA e){
     ….  }
  }
}
```

```
public ClaseC{
  public int m3(..) throws ErrorA {
    ….
  }
}
```

Flujo de Programa `try-throw-catch`

- Bloque Try
  - Las sentencias encerradas en el bloque Try son las sentencias protegidas (bloque protegido).
  - En el método metodoAux, si la condición es true, se lanza la excepción
    - Se corta la ejecución de metodoAux, y el control pasa al bloque catch después del bloque try
  - Si la condición es false
    - La excepción no se lanza, el método se ejecuta con normalidad
    - Las sentencias restantes en el bloque try (aquellas que siguen el throw condicional) son ejecutadas

- Bloque Catch
  - Se ejecuta si una excepción es lanzada. Es el bloque manejador de la excepción
  - Puede terminar la ejecución con una sentencia exit (aborta el programa)
  - Si no hace exit, la ejecución se reanuda después del bloque catch

- Las sentencias después del bloque Catch se ejecutan tanto si la excepción fue lanzada o no

Ejemplo de manejo de excepciones

bloque `try`

sentencia `throw` en el método dispara la excepción

bloque `catch`

```
/** donas por vaso de leche */

int contDonas=0, contLeche=0;
    double donasPorVaso=0.0;

    try
    {
        System.out.println("Ingrese el nro. de donas:");
        contDonas = TecladoIn.readLineInt( );

        contLeche = ingresarLeche( );

        donasPorVaso = (double)donasPorVasos/(double)contLeche;
        System.out.println(contDonas + " donas.");
        System.out.println(contLeche + " vasos de leche.");
        System.out.println(" Hay " + donasPorVaso
                           + " donas por cada vaso de leche.");
    }

    catch(Exception e)
    {
        System.out.println(e.getMessage( ));
        System.out.println(" Ir a comprar leche.");
    }

    System.out.println(" Fin del programa.");
```

Ejemplo de manejo de excepciones

```
/** donas por vaso de leche */
int contDonas=0, contLeche=0;
        double donasPorVaso=0.0;

        try
        {
             System.out.println("Ingrese el nro. de donas:");
             contDonas = TecladoIn.readLineInt( );
              contLeche = ingresarLeche();

                  dona
                  Syst
                  Syst
                  Syst
        }

        catch(Exception e)
        {
             System.out.println(e.getMessage( ));
             System.out.println(" Ir a comprar leche.");
        }

        System.out.println(" Fin del programa.");
```

```
public double ingresarLeche()  throws Exception{
   System.out.println(" Ingrese el nro.de vasos de
leche:");
   contLeche = TecladoIn.readLineInt( );

   if (contLeche < 1)
      throw new Exception("Excepcion: No hay leche!");
}
```

bloque `try`

sentencia `Throw`

dispara la excepción

bloque `catch`

Más acerca del Bloque `catch`

- **Exception** es la clase base de todas las excepciones
- El bloque catch no es una definición de método (aunque parece similar)
- Cada excepción hereda el método **getMessage**
  - Este método carga el string dado al objeto-excepción cuando fue lanzada la excepción, ej.
  - throw new Exception("Mensaje cargado");
- Un bloque **catch** se aplica sólo sobre el bloque **try** que inmediatamente lo precede
- Si ninguna excepción es lanzada, el bloque catch es ignorado

Definiendo clases de excepción propias

```
public class ExcepcionDividePorCero extends Exception
{
    public ExcepcionDividePorCero ()
    {
        super("Dividiendo por Cero!");
    }

    public ExcepcionDividePorCero (String mensaje)
    {
        super(mensaje);
    }
}
```

- Extiende (hereda) la clase Exception ya definida
- El único método que necesitamos definir es el constructor
  - Incluye un constructor que toma un argumento `String`
  - También un constructor por defecto con un mensaje string por defecto

Usando la clase **ExcepcionDividePorCero**

```
public void hacerEsto( ) {
try
   {
    System.out.println("Ingrese numerador:");
    this.numerador = TecladoIn.readLineInt( );
    System.out.println("Ingrese denominador:");
    this.denominador = TecladoIn.readLineInt( );
    if (this.denominador == 0)
    throw new ExcepcionDividePorCero("Error:Division por 0" );
    double cociente =
       (double)this.numerador/(double)this.denominador;
    System.out.println(this.numerador + "/" +
              this.denominador + " = " + cociente);
    }
catch(ExcepcionDividePorCero e)
    {
    System.out.println(e.getMessage( ));
    System.out.println("El calculo no fue realizado");
    }
}
```

Resumen

- Una excepción es un objeto descendiente de la clase Exception
- El manejo de excepciones permite diseñar código para los casos normales separados de los casos excepcionales
- Podemos usar las clases de excepción predefinidas o definir la nuestra
- Las excepciones pueden ser lanzadas por:
  - Ciertas sentencias Java
  - Los métodos de las librerías de clase
  - Un bloque try
  - Una definición de método sin bloque try, pero la invocación al método está ubicada dentro de un bloque try

Resumen II

- Una excepción es atrapada por un bloque catch
- Un bloque try puede estar seguido por más de un bloque catch
- Más de un bloque catch puede ser capaz de manejar la excepción
  - El primer bloque catch que pueda manejar la excepción, es el único que se ejecuta
  - Colocar los bloques catch en orden de especificidad, y el más general al final
- No exagerar el uso de excepciones
  - Reservarlo para situaciones donde no se pueda resolver de otra manera

ErrorTDA

- Podemos crear una clase ErrorTDA para avisar cuando falla algo en un método de cualquier TDA definido por nosotros
- Para predefinir los mensajes podemos usar un código
- Errores más comunes:
  - 1: Datos de entrada inválidos
  - 2: Objeto inmutable: no se puede modificar
  - 3: Elemento no encontrado
  - 4: Elemento repetido
  - 5: Posición inválida
  - Etc.
- El método getMensaje tiene un switch y asigna el mensaje según el código del error

Una clase ErrorTDA

ErrorTDA (extiende Exception)
- codigo

CONSTRUCTOR
+ ErrorTDA (int codError);
 	
OBSERVADOR
+ getCodigo(): int
//devuelve el codigo del mensaje
+ getMensaje(): String
//devuelve el mensaje correspondiente al
código ingresado
 	
MODIFICADOR
// no tiene modificadores (es inmutable)

Mensajes de ErrorTDA

```
// Mensajes generales para TDAs
        case 1:   mensaje = "Datos de entrada invalidos";
        case 2:   mensaje = "Objeto inmutable, no se puede modificar";
        case 3:   mensaje = "La operacion no termino exitosamente";

// Mensajes para manejo de estructuras dinamicas
        case 4:   mensaje = "Posicion no valida";
        case 5:   mensaje = "Elemento no encontrado";
        case 6:   mensaje = "Elemento o clave repetido";
        case 7:   mensaje = "Estructura llena";
        case 8:   mensaje = "Estructura vacia";

// Mensajes para manejo de archivos
        case 100: mensaje = "Error abriendo el archivo";
        case 101: mensaje = "Error de escritura en archivo";
        case 102: mensaje = "Error de lectura de archivo";
        case 103: mensaje = "Error cerrando Archivo";
        case 104: mensaje = "Archivo inexistente";

// Mensaje con codigo invalido
        default:  mensaje = "Error indeterminado";
```

Ejemplo: dividir enteros

- dividir(Entero):Entero
  - Si el valor pasado por parámetro es 0, debe disparar una excepción

```
public class Entero{

  public Entero dividir(Entero e) throws ErrorTDA {
     if (e.igualCero())
        throw new ErrorTDA(1); //dato entrada invalido
     else
        return ...
  }
}
```

Ejemplo (cont)

```
public class TestEntero{

  public static void main(String[] arg){

    Entero e1 = new Entero(7);
    Entero e2 = new Entero(0);
    try {
      e3 = e1.dividir(e2);   // puede disparar exception ErrorTDA
      System.out.println("Resultado: " + e3.aCadena());
    }
    catch (ErrorTDA e){
      System.out.println("Horror: " + e.getMensaje());
    }
  }
}
```

Resumen

- La validación de los datos enviados a un TDA es esencial
- El programa llamador es responsable de chequear los datos (que pueda) al invocar un método de un TDA
  - Especialmente cuando los datos se lean por teclado (el usuario suele cometer muchos errores involuntarios)
- Por las dudas, el TDA debe chequear los datos que reciba y avisar al programa que lo ha invocado
  - Una manera apropiada para que se comunique el programa llamador y el TDA es utilizar una excepción (para eso creamos ErrorTDA)
  - El programa llamador debe capturar la excepción y actuar en consecuencia o emitir mensajes de error apropiados