



Departamento de Programación  
Facultad de Informática  
Universidad Nacional del Comahue



# Programación Concurrente



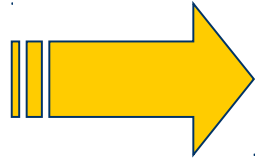
*Presentación y Repaso*

# Temario

- Herencia en Java
- Visibilidad en Java
- Interfaces
- **Vectores**
- Wrappers
- Excepciones

# Interfaces de colecciones

- Vectores



Mejor utilizar otras colecciones: List, Map,

¿por qué?

# Iteration

## Iteration (Interfaz)

- permiten recorrer una colección de principio a fin
- *Iterator* es una interfaz con 3 operaciones
  - *hasNext*
  - *next* (retorna el elemento actual y avanza la posición)
  - *remove* (remueve el último elemento que fue retornado por next)

# Uso de Iterator

```
int cuenta(Iterator e, Object obj) {  
    int c=0;  
    while (e.hasNext())  
        if (e.next ().equals(obj)) c++;  
    return c;  
}  
...  
Vector v= new Vector();  
Hashtable h = new Hashtable();  
Empleado emp = new Empleado("Juan");  
...  
int a = cuenta(v.iterator(), emp);  
int b = cuenta(h.iterator(), emp);
```

# Características

- No mejoran la performance sino que sirven para una **mejor abstracción**
- No se necesita saber nada de la estructura para utilizar un **Iterator**.
- **Vector** es reemplazados por **ArrayList** en versiones posteriores de Java.

# Por qué existen los elementos Genéricos?

- Al seleccionar un elemento de una colección, se debe convertir al tipo de elemento que se almacena en dicha colección.
- El compilador no comprueba que el **cast** sea del mismo tipo de la colección, por lo que el **cast** puede fallar en tiempo de ejecución.
- Los genéricos proporcionan una forma de determinar el tipo de una colección para el compilador, por lo que se puede comprobar

# Utilización de genéricos

- Eliminar las palabras de 4 letras de una colección cualquiera

```
static void eliminar(Collection c) {  
    for (Iterator i = c.iterator(); i.hasNext(); )  
        if (((String) i.next()).length() == 4)  
            i.remove();  
}
```

*Si utilizamos genéricos:*

```
static void eliminar(Collection<String> c) {  
    for (Iterator<String> i = c.iterator(); i.hasNext(); )  
        if (i.next().length() == 4)  
            i.remove();  
}
```



# Usos de Enumeration e Iterator

```
Vector<String> v=new Vector<String>();  
v.add("Ana");      v.add("Betina");      v.add("Carlos");  
v.add("Dora");      v.add("Eduardo");      v.add("Fernando");
```

```
Iterator<String> it=v.iterator();  
while(it.hasNext()) {  
    String value=(String) it.next();  
    System.out.println(value);  
}
```

# Temario

- Herencia en Java
- Visibilidad en Java
- Interfaces
- Vectores
- Wrappers
- Excepciones

# Clases envoltorio (Wrapper)

- Envuelven tipos primitivos en una estructura de clases
- Todos los tipos primitivos tienen su tipo wrapper
- La clase incluye constantes y métodos estáticos y conversores de y hacia los tipos primitivos

Tipo Primitivo	Tipo Clase	Metodo de conversión
int	Integer	<b>intValue()</b>
long	Long	<b>longValue()</b>
float	Float	<b>floatValue()</b>
double	Double	<b>doubleValue()</b>
char	Character	<b>charValue()</b>

# Clase wrapper: Integer

- Cómo se declara?

```
Integer n = new Integer();
```

- Cómo pasa de un objeto a una primitiva?

```
int i = n.intValue(); //intValue retorna un int
```

- Constantes de la clase **Integer**:

```
Integer.MAX_VALUE      //valor máximo y mínimo
```

```
Integer.MIN_VALUE
```

- Métodos de **Integer**:

```
Integer.valueOf("123") convierte un String en Integer
```

```
Integer.toString(123) convierte un Integer en String
```

# Uso de las clases Wrapper

## Diferencias en el uso de clases wrapper y tipos primitivos

### Clase Wrapper

- declaración:  
`Integer n;`
- La variable `n` contendrá una referencia a un objeto que contendrá un entero
- declaración e inicialización:  
`Integer n = new Integer(0);`
- asignación:  
`n = new Integer(5);`

### Tipo Primitivo

- declaración:  
`int n;`
- La variable `n` contendrá un valor entero
- declaración e inicialización:  
`int n = 0;`
- asignación:  
`n = 5;`

# Crear Clases Wrapper

- Para declarar y crear una variable de un tipo de clase wrapper se usa uno de los constructores
- Ejemplos para clase `Integer`
  - Constructor Vacío  
`Integer n = new Integer();`
  - Constructor con argumento del tipo primitivo asociado (en este caso `int`)  
`Integer n = new Integer(5);`
  - Constructor con argumento `String`  
`Integer n = new Integer("5");`

# Usar Clases Wrapper

- Cada clase incluye métodos útiles para el tipo.

Ejemplos:

- **Convertir String en tipo numérico**

```
int j = Integer.parseInt("7");
```

- **Convertir clase wrapper en el tipo primitivo correspondiente**

```
Integer i = new Integer("5");  
int k = i.intValue();
```

Tipo Primitivo	Tipo Clase	Método estático. Convierte String en tipo primitivo	Método para reconvertir al tipo prim.
int	Integer	parseInt(String s)	intValue()
long	Long	parseLong(String s)	longValue()
float	Float	parseFloat(String s)	floatValue()
double	Double	parseDouble(String s)	doubleValue()
boolean	Boolean	parseBoolean(String s)	booleanValue()
char	Character		charValue()

# Constantes en clases Wrapper

- También tienen constantes útiles:
  - Clase Integer
    - **Integer.MAX\_VALUE**: valor integer máximo que la computadora puede representar
    - **Integer.MIN\_VALUE**: valor integer mínimo que la computadora puede representar
  - Clase Double
    - **Double.MAX\_VALUE**: valor integer máximo que la computadora puede representar
    - **Double.MIN\_VALUE**: valor integer mínimo que la computadora puede representar
  - Idem Float y Long



# Método toString

- Además, todas las clases wrapper tienen un método estático `toString` que recibe un valor del tipo primitivo y devuelve una cadena
- Ejemplos
  - `Double.toString(199.98)` devuelve “199.98”
  - `Boolean.toString(false)` devuelve “false”
  - `Character.toString('h')` devuelve “h”
  - Idem para Integer, Long y Float

# Temario

- Herencia en Java
- Visibilidad en Java
- Interfaces
- Vectores
- Wrappers
- Excepciones

# Validando datos de entrada

- Vimos que las aserciones no son buen mecanismo para validar los datos de entrada de un método público o programa
- Las aserciones, cuando no se cumplen, cortan el programa
  - Por eso las usamos para testing (antes de entregar el programa al usuario)
- ¿Qué otro mecanismo tenemos para detectar errores y que el programa pueda restablecerse?

# ¿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

- Un programa **correcto** es aquel que actúa de acuerdo a su **especificación**.
- Un programa **confiable** es correcto y además tiene un comportamiento previsible, es decir actúa razonablemente no sólo en situaciones normales sino también en circunstancias anómalas, como por ejemplo fallas de hardware.
- Desde el punto de vista de la aplicación las situaciones consideradas normales dependen del diseñador que analiza el problema.

# Excepciones

- Una excepción es un **evento anormal** durante la ejecución que puede provocar que una operación falle.
- 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".

# 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.



# Excepciones

- Ejemplos típicos de excepciones detectadas implícitamente son:
  - **ArithmeticException** División por 0, señalado por la operación /
  - **ArrayIndexOutOfBoundsException** El acceso fuera de rango dentro de un arreglo, señalado 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.

# Excepciones

- En los ejemplos anteriores 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.

# Excepciones

- 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 mantenimiento, pero no se debe abusar de ellas.

# 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

# Warning (Precaución)



- Los ejemplos que veremos a continuación están simplificados con fines educativos.
- Los programas reales son más complicados y generalmente tienen una organización diferente

# La terna **try-throw-catch**

## Organización básica del código

```
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");
```

...

# 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

```
/** 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.");
```

bloque try

sentencia throw en  
el método  
dispara la excepción

bloque  
catch



# Ejemplo de manejo de excepciones

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

bloque try

sentencia Throw  
dispara la  
excepción

bloque  
catch

```
try  
{
```

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

```
public double ingresarLeche() {
```

```
    System.out.println(" Ingrese el nro.de vasos de  
    leche:");  
    contLeche = TecladoIn.readLineInt( );
```

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

```
catch(Exception e)
```

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

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

Si el usuario ingresa un número positivo para el número de vasos, entonces ninguna excepción es lanzada.

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

    donasPorVaso = (double)contDonas/(double)contLeche;


    System.out.println(contDonas + " donas.");
    System.out.println(contLeche + " vasos de leche.");
    System.out.println("Hay " + donasPorVaso);
}
catch (Exception e)
{
    System.out.println(e.getMessage());
    System.out.println("Ir a comprar leche.");
}
System.out.println("Fin del Programa.");
```

No se ejecuta porque no se dispara la excepción

Si el usuario ingresa un cero o un número negativo de vasos, se dispara una excepción.

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

    donasPorVaso = (double)contDonas / (double)contLeche;
    System.out.println(contDonas + " donas.");
    System.out.println(contLeche + " vasos de leche.");
    System.out.println("Hay " + donasPorVaso);
}
catch (Exception e)
{
    System.out.println(e.getMessage());
    System.out.println("Ir a comprar leche.");
}
System.out.println("Fin del Programa.");
```



No se ejecuta porque se disparó la excepción

# 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 = ingresarDenominador( );  
        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");  
    }  
}
```

# Usando la clase **ExcepcionDividePorCero**

```
public double ingresarDenominador( ) {  
    System.out.println("Ingrese denominador:");  
    double denominador = TecladoIn.readLineInt( );  
    if (denominador == 0)  
        throw new ExcepcionDividePorCero  
            ("Error:Division por 0" );  
    else return denominador  
}
```

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

- Un método puede lanzar más de una excepción
- Los bloques **catch** inmediatamente 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 especificidad: 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  
    }
```



# El Bloque **finally**

- Se puede agregar un bloque **finally** después de los bloques try/catch
- El bloque **finally** se ejecuta sin importar si el bloque catch se ejecuta
- La organización del código utilizando el bloque finally será:

```
try {bloque}
catch (...) {bloque}
finally
{
    <Código a ejecutarse se dispare o no una
    excepción>
}
```

# Tres Posibilidades para un bloque **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.
- 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.
- 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.

# 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 comuniquen 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