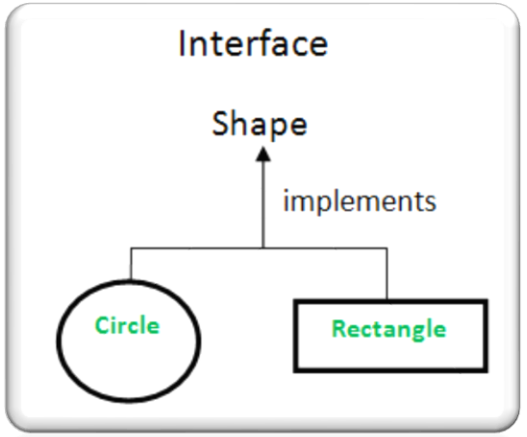
[Estructuras de Datos]



CONCEPTOS BÁSICOS SOBRE EL MANEJO DE EXCEPCIONES.

GENERICIDAD PARAMÉTRICA

INTERFACES.

Copyright

Recursos adicionales

Enlaces externos de interés

- Acceda a información útil mediante enlaces externos a:
  - **código fuente** disponible de forma **online**.
  - **otro material** disponible de forma **online**.

CONCEPTOS BÁSICOS SOBRE MANEJO DE EXCEPCIONES

¿Qué son las excepciones?

- Las excepciones son **eventos inesperados** que ocurren durante la **ejecución** de un programa.
  - Puede ser el resultado de una **condición de error** o una **entrada incorrecta**.
- En Programación Orientada a Objetos (POO), las excepciones son **objetos lanzados** por un código que **detecta** una **condición inesperada**, y es **capturada** por **otro código** que debe **reaccionar** ante esta situación **intentando repararla**.

Tipos de excepciones

CHECKED

Son excepciones que **obligan** tanto al **servidor** como al **cliente** de un servicio a **tomar medidas** frente a estas.

La clase **servidor** debe **indicar** en la **signatura** del método o constructor que podría lanzar este tipo de excepciones.

La clase **cliente**, al utilizar un **servicio** que puede **lanzar excepciones** de este tipo, debe **capturarlas** y **reaccionar** ante estas, o bien, debe **propagarlas** indicando esto en la signatura del método o constructor que la propaga.

UNCHECKED

Son excepciones que **no obligan** ni al **servidor** ni al **cliente** de un servicio a **tomar medidas** frente a estas.

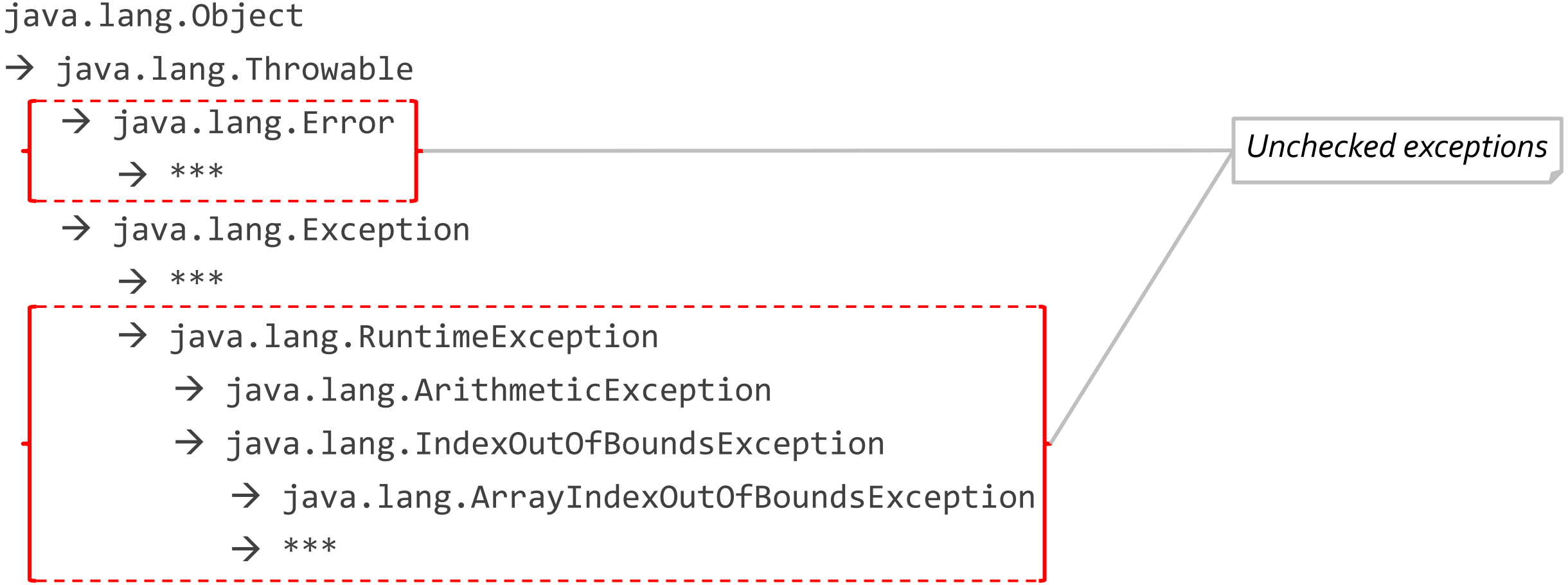
La clase **servidor no debe** indicar en la **signatura** de un método o constructor que podría lanzar este tipo de excepciones.

La clase **cliente**, al utilizar un **servicio** que puede **lanzar excepciones** de este tipo (pero que no lo especifica en la signatura del método o constructor), **no está obligado** a **capturarlas** y **reaccionar** ante estas.

Tipos de excepciones :: jerarquía de herencia

java.lang.Object
→ java.lang.Throwable
  → java.lang.Error
    → ***
  → java.lang.Exception
    → ***
    → java.lang.RuntimeException
      → java.lang.ArithmeticException
      → java.lang.IndexOutOfBoundsException
        → java.lang.ArrayIndexOutOfBoundsException
        → ***

Tipos de excepciones :: jerarquía de herencia



Tipos de excepciones :: jerarquía de herencia

- java.lang.Object
  - → java.lang.Throwable
    - → java.lang.Error
      - → ***
    - → java.lang.Exception
      - → ***
      - → java.lang.RuntimeException
        - → java.lang.ArithmeticException
        - → java.lang.IndexOutOfBoundsException
          - → java.lang.ArrayIndexOutOfBoundsException
          - → ***

Ante divisiones por cero.

Ante el acceso a una posición inválida dentro de un arreglo.

Tipos de excepciones :: jerarquía de herencia

```
java.lang.Object
→ java.lang.Throwable
    → java.lang.Error
        → ***
    → java.lang.Exception
        → ***
        → java.lang.RuntimeException
            → java.lang.ArithmeticException
            → java.lang.IndexOutOfBoundsException
                → java.lang.ArrayIndexOutOfBoundsException
                → ***
```

Checked exceptions

Tipos de excepciones :: jerarquía de herencia

- java.lang.Object
  - → java.lang.Throwable
    - → java.lang.Error
      - → ***
    - → java.lang.Exception
      - → ***
      - → java.lang.RuntimeException
        - → java.lang.ArithmeticException
        - → java.lang.IndexOutOfBoundsException
          - → java.lang.ArrayIndexOutOfBoundsException
          - → ***

En este punto de la jerarquía definiremos las excepciones que utilizaremos en todos los TDAs de la materia.

En esta materia solamente **se permitirá** el uso de excepciones del tipo **checked**.

Tipos de excepciones: unchecked

- Considere el siguiente fragmento de código:

```
public class Calculadora {
  public static float dividir(int dividendo, int divisor){
    return dividendo/divisor;
  }

  public static void main(String [] args){
    float resultado = dividir(10, 0);
    System.out.println("El resultado es: " + resultado);
  }
}
```

- El **compilador** no indica problema alguno.
- En ejecución, al **intentar dividir 10 por 0,** se **provoca** una **situación excepcional**, que Java modela mediante una **ArithmeticException**.
- Note que esta excepción, al no ser capturada, hace que el programa en ejecución **aborte**.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Calculadora.Calculadora.dividir(Calculadora.java:6)
at Calculadora.Calculadora.main(Calculadora.java:10)
```

Tipos de excepciones: checked

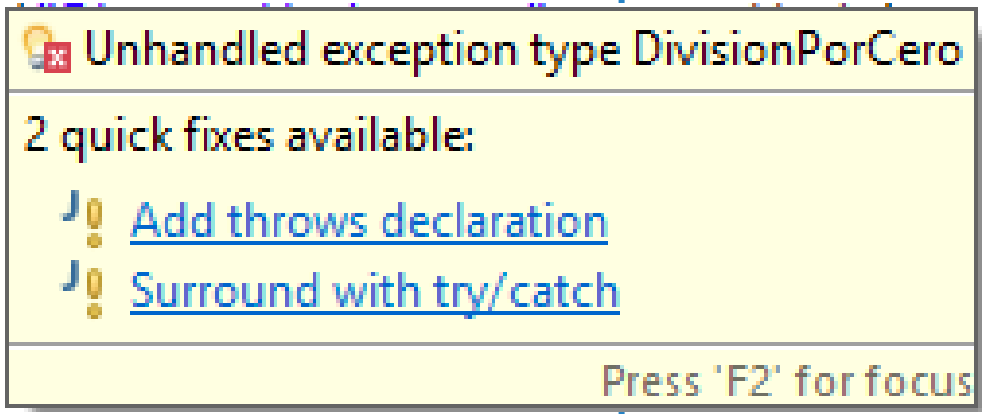
- Considere ahora que al fragmento de código anterior, se le **agrega funcionalidad** para que el **método dividir chequee** que el **divisor** sea **distinto** de **cero,** y en caso contrario, **lance** una **excepción** notificándole al **cliente** dicho **error**:

```
public class Calculadora {
  public static float dividir(int dividendo, int divisor) throws DivisionPorCero{
    if (divisor == 0) throw new DivisionPorCero("Error");
    return dividendo/divisor;
  }
}
```

- Note que ahora la clase **servidor**, **especifica** en su **signatura** que puede **lanzarse** una **excepción** del tipo **DivisionPorCero** (es una checked exception).
- Cualquier clase **cliente** que utilice el método **dividir(…)**, está obligada a tomar medidas frente a esta situación.
  - Puede **capturar** la **excepción**, o bien, **propagarla**.

Tipos de excepciones: checked

```
public class Calculadora {
  public static float dividir(int dividendo, int divisor) throws DivisionPorCero{
    if (divisor == 0) throw new DivisionPorCero("Error");
    return dividendo/divisor;
  }

  public static void main(String [] args){
    float resultado = dividir(10, 0);
    System.out.println("El resultado es: " + resultado);
  }
}
```



- Tal cual está el código, el **compilador** indica un **error** en la llamada al **método dividir(…)**.
- **Eclipse** nos permite **posicionarnos** por sobre la línea de **error**, y nos ofrece un conjunto de soluciones rápidas disponibles.
  - **Capturar** la **excepción** con **Try-Catch**
  - **Propagar** la **excepción**.

Tipos de excepciones: checked

- **Capturar** la **excepción** con **Try-Catch**

```
public static void main(String [] args){
  float resultado = 0;
  try {
    resultado = dividir(10, 0);
  } catch (DivisionPorCero e) {
    System.out.println(e.getMessage());
  }
  System.out.println("El resultado es: " + resultado);
}
```

¿Qué sucederá cuando se captura la excepción? ¿Y cuando la excepción es propagada?

- **Propagar** la **excepción**.

```
public static void main(String [] args) throws DivisionPorCero{
  float resultado = dividir(10, 0);
  System.out.println("El resultado es: " + resultado);
}
```

Tipos de excepciones: checked

- **Capturar** la **excepción** con **Try-Catch**

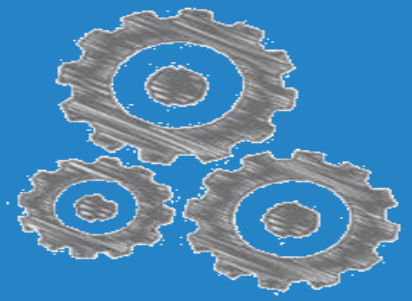
```
Error
El resultado es: 0.0
```

- **Propagar** la **excepción**.

```
Exception in thread "main" Calculadora.DivisionPorCero
at Calculadora.Calculadora.dividir(Calculadora.java:5)
at Calculadora.Calculadora.main(Calculadora.java:10)
```

¿Qué sucederá cuando se captura la excepción? ¿Y cuando la excepción es propagada?

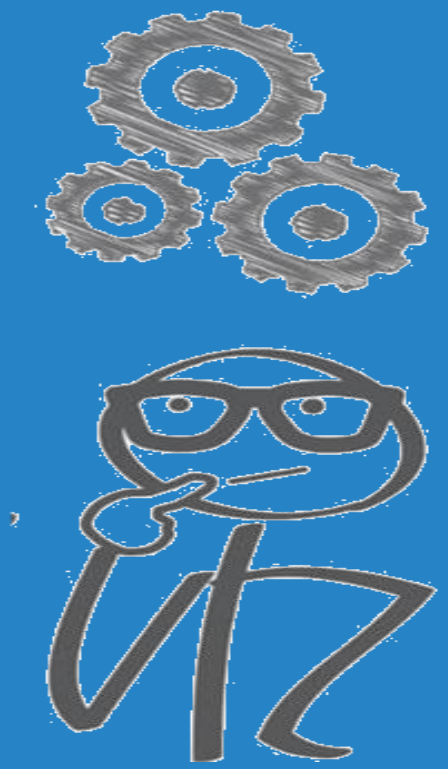
Problema para analizar…



Manejo de excepciones

- Imagine un sistema para modelar **transacciones bancarias**, en donde dos clientes (**CltA** y **CltB**), van a **transferirse** dinero entre sí. La clase **Banco** dispone de un método **transferir(…)** que recepciona dos clientes, y una cantidad de dinero a transferir desde la cuenta del primer cliente hacia la cuenta del segundo cliente.
- La clase **Cliente** cuenta con los métodos **extraer**(…) en donde se indica cuánto dinero extraerle al cliente, y un método **depositar(…)** que indica la cantidad de dinero a incrementarle al mismo.
- Estos métodos lanzan las excepciones **SinFondosException** y **SuperaMaximoException** respectivamente, en el caso que al **extraer no existan fondos** en la cuenta, o cuando al momento de **depositar,** el **dinero que se acumula** en la cuenta **supera** el **máximo permitido** para el cliente en cuestión.

Problema para analizar…



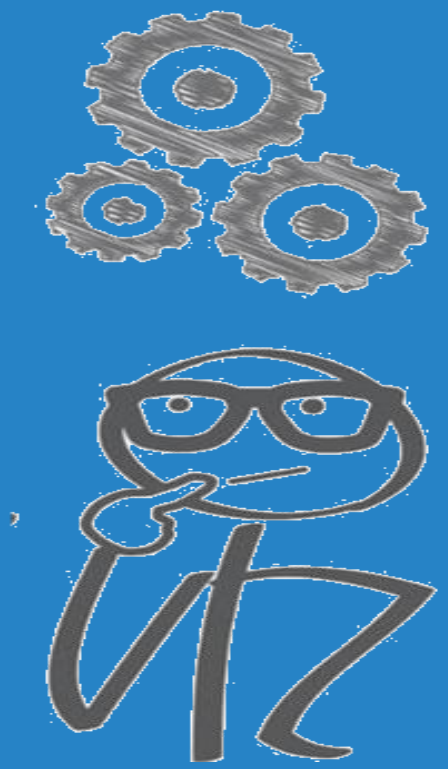
Manejo de excepciones

- Dada la siguiente implementación de **transferir(…)** en la clase **Banco**, ¿qué **errores** en cuanto a la **funcionalidad** detecta?

```
public class Banco {
  public void transferir(Cliente CltA, Cliente CltB, float monto){
    try {
      CltA.extraer(monto);
    } catch (SinFondosException e) {
      System.out.println(e.getMessage());
    }
    try {
      CltB.depositar(monto);
    } catch (SuperaMaximoException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

Teniendo en cuenta la operatoria usual en un banco: ¿qué sucede si la operación **extraer** falla? ¿Se **deposita** de igual forma?
De igual forma a la inversa: ¿qué sucede si la operación **depositar** falla? ¿La operación **extraer** no se debe anular?

Problema para analizar…



Manejo de excepciones

- Dada la siguiente implementación de **transferir(…)** en la clase **Banco**, ¿qué **errores** en cuanto a la **funcionalidad** detecta?

```
public class Banco {
  public void transferir(Cliente CltA, Cliente CltB, float monto){
    try {
      CltA.extraer(monto);
    } catch (SinFondosException e) {
      System.out.println(e.getMessage());
    }
    try {
      CltB.depositar(monto);
    } catch (SuperaMaximoException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

No siempre una **excepción** puede **manejarse imprimiendo** un **mensaje**. Por el contrario, ante estas situaciones **se deben tomar** las **medidas necesarias** para **garantizar** la **correctitud** y **robutez** de un programa.

GENERICIDAD PARAMÉTRICA EN JAVA

Repaso: Genericidad Pre Java 5

- Una alternativa para admitir **colecciones genéricas** en las versiones de Java anteriores a la 5, era mediante el uso de **genericidad** por **polimorfismo**.
  - Por ejemplo, declarando un **arreglo** de elementos **Object**:

```
private Object [] coleccion = new Object[10];
```

- Naturalmente, esto presentaba serias dificultades en el manejo de los datos:
  - No se puede garantizar homogeneidad de los elementos.
  - Se deben realizar castings explícitos a la hora de recuperar elementos.

```
coleccion[0] = new Integer(10);
coleccion[1] = new String("Hola que tal");

Integer a = (Integer) coleccion[0];
String b = (String) coleccion[1];
```

Genericidad Paramétrica: Java 5 y posterior

- En las versiones de Java 5 y posteriores, se incluye **genericidad paramétrica**.
  - Un tipo **genérico** es un **tipo de dato** que es definido en **tiempo de ejecución**, pero que **permite** definir una clase en función de **parámetros formales de tipo** que **abstraen** el tipo de dato interno que en ejecución será instanciado.

```
public class Encapsulador <E> {
  private E elemento;

  public E getElemento(){ return elemento;}
  public void setElemento(E e){ elemento = e;}
}
```

- Note que la clase **Encapsulador**, está totalmente definida en términos de un **tipo de dato E** que se **desconoce** en tiempo de **compilación** (puede ser cualquier tipo de dato).
- La clase **almacena** un elemento **E**, **retorna** un elemento **E**, **modifica** un elemento **E**, lo que permite **verificar** la **correctitud** de **tipos** en todo momento.

Genericidad Paramétrica: Java 5 y posterior

- En las versiones de Java 5 y posteriores, se incluye **genericidad paramétrica**.
  - Un tipo **genérico** es un **tipo de dato** que es definido en **tiempo de ejecución**, pero que **permite** definir una clase en función de **parámetros formales de tipo** que **abstraen** el tipo de dato interno que en ejecución será instanciado.

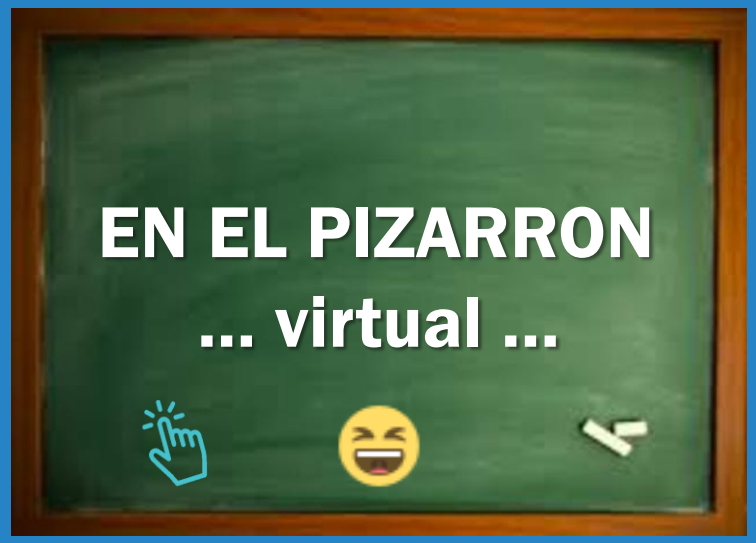
```
public class Encapsulador <E> {
  private E elemento;

  public E getElemento(){ return elemento;}
  public void setElemento(E e){ elemento = e;}
}
```

- Note que en **tiempo de ejecución**, el tipo genérico **E** fue instanciado con **Integer** y **Float** para las clases referenciadas por **encapInt**, **encapFloat** respectivamente.

```
public static void main(String [] arg){
  Encapsulador<Integer> encapInt;
  Encapsulador<Float> encapFloat;

  encapInt = new Encapsulador<Integer>();
  encapInt.setElemento(10);

  encapFloat = new Encapsulador<Float>();
  encapFloat.setElemento(10.5f);

  System.out.println("Entero: " + encapInt.getElemento());
  System.out.println("Float: " + encapFloat.getElemento());
}
```

INTERFACES EN JAVA

Problema propuesto.



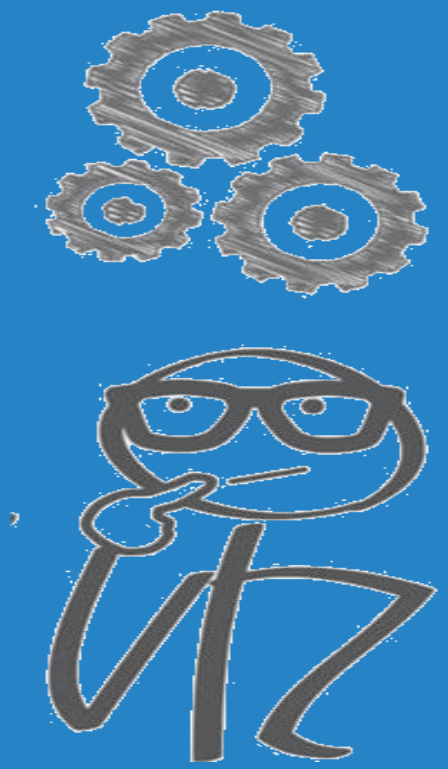
Colección de elementos

- Defina una interfaz **colección de elementos genéricos** que permita las siguientes operaciones:
  - void insertar(elemento e)
  - boolean pertenece(elemento e)
  - void eliminar(elemento e)
  - void mostrar()

- Desarrolle una clase **ColeccionConArreglo**, que implemente una colección de elementos genérica, almacenando internamente los elementos en un **arreglo**.

- Desarrolle una clase **ColeccionConVector**, que implemente una colección de elementos genérica, almacenando internamente los elementos en un **vector**.

Para analizar…



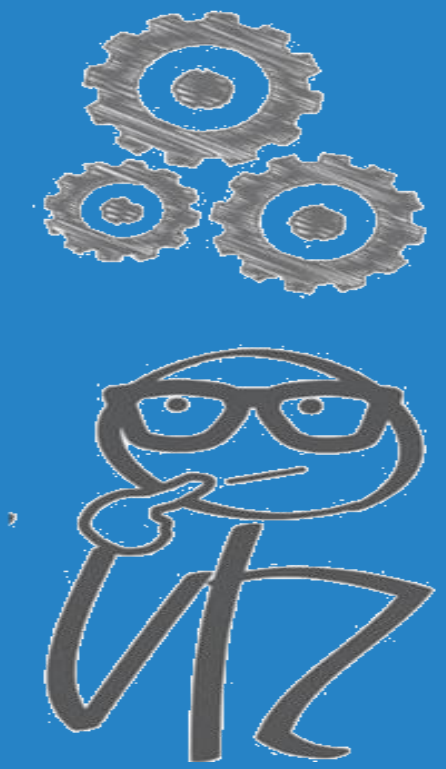
Colección de elementos

- ¿Qué **ventaja** existe desde el punto de vista del **cliente** el haber definido una **interfaz** y que esta, a la vez, haya sido **genérica**?
- ¿Qué hubiese sucedido si en lugar de definir una **interfaz genérica** colección, se hubiesen implementado las clases **ColeccionEnteros** y **ColeccionReales** directamente?

```
public static void main(String [] args){
  Coleccion<Integer> col = ColeccionConArreglo<Integer>();

  col.insertar(10);
  col.insertar(15);

  if (col.pertenece(15)) col.eliminar(15);
  else
    System.out.println("El numero 15 no pertenece a la colección");
}
```

Por un lado, el haber definido una interfaz **Colección genérica**, nos permite que **col** pueda ser usado también, por ejemplo, para almacenar Floats. Por ejemplo, es posible definir e instanciar **col** de la siguiente forma, sin modificar el código restante : `Coleccion<Float> col = ColeccionConArreglo<Float>();`

Para analizar…



Colección de elementos

- ¿Qué **ventaja** existe desde el punto de vista del **cliente** el haber definido una **interfaz** y que esta, a la vez, haya sido **genérica**?
- ¿Qué hubiese sucedido si en lugar de definir una **interfaz genérica** colección, se hubiesen implementado las clases **ColeccionEnteros** y **ColeccionReales** directamente?

```
public static void main(String [] args){
  Coleccion<Integer> col = ColeccionConArreglo<Integer>();

  col.insertar(10);
  col.insertar(15);

  if (col.pertenece(15)) col.eliminar(15);
  else
    System.out.println("El numero 15 no pertenece a la colección");
}
```

Por otro lado, el haber definido una interfaz **Colección** permite que **col** pueda ser usado como una **colección** **sin** **importar** cómo realmente está **implementada**. Por ejemplo, es posible instanciar **col** de la siguiente forma, sin modificar el código restante: `colInt = ColeccionConVector<Integer>();`

Para tener en cuenta…

Tipos de datos abstractos (TDA)

- En Estructuras de Datos, nos basaremos en el concepto de **interfaces** para definir **tipos de datos**, y **estudiaremos** las diferentes **alternativas existentes** para **implementar** dichos **tipos de datos**.

- Es importante tener en cuenta que el TDA es quien **define** el **conjunto de datos** y las **operaciones** que dicho tipo admite.
  - En el ejemplo anterior, indicamos que una **Colección** almacena elementos genéricos e, independientemente de cómo esté implementada, permite **insertar**, **eliminar**, consultar **pertenencia** y **mostrar** los elementos que almacena.

Para tener en cuenta…

Tipos de datos abstractos (TDA)

- De una forma más informal, un **TDA** define un **conjunto de datos** y qué se puede **hacer** con dichos datos.

- Las **implementaciones** de estos **TDA**, nos dicen cómo es que los **datos** se **organizan**, y cómo las **operaciones** se **llevan a cabo**.
  - En el ejemplo anterior, vimos **dos implementaciones** de una **colección** utilizando internamente un **arreglo** o un **vector** respectivamente.

- En la materia **estudiaremos** diferentes **implementaciones** para varios TDAs. El análisis se centrará en las performances de estas implementaciones en cuanto al espacio en memoria y tiempo de ejecución requerido por sus operaciones.

Fin de la presentación.