fiuba

algo3

Temas varios de POO

(profundización)

Pablo Suárez

psuarez@fi.uba.ar

Contexto

- Vemos que Pharo nos permite definir métodos y atributos “class side”
  - Pero no sabemos para qué sirven
- Venimos usando UML desde el principio
  - Pero nos falta formalizar
- Trabajamos con excepciones desde que vimos contratos
  - Pero no profundizamos en el tema en sí
- Las clases se agrupan en paquetes
  - Pero no sabemos cuál es su semántica
  - ¿Depende del lenguaje?

Temas varios de POO

Temario

Atributos y métodos de la clase
Inicializando objetos
Excepciones
Chequeo estático de excepciones
Diagrama de estados
Cierre de UML
Colecciones e iteradores
Genericidad

A modo de cierre de POO

Inicialización en Java

Hay constructores

```
Date d = new Date();
```

El constructor puede tener parámetros

```
Cuenta c = new CajaAhorro(1234, "Ana");
```

Debería dejar al objeto en un estado válido

=> debe cumplir con los invariantes

Si no hay constructor en una clase, el compilador crea uno por defecto

Ojo: sin parámetros

Inicialización en Smalltalk (1)

No hay constructores

- sólo está el método de clase *new*

Debería dejar al objeto en un estado válido

- => debe cumplir con los invariantes
- El *new* no es seguro

Hay un método *initialize*, que se puede redefinir

- De todas maneras, no tiene parámetros
  - => Tampoco es seguro

Inicialización en Smalltalk (2)

Por eso definimos un método:

```
CuentaBancaria >>
    inicializarConNumero: numero conTitular: titular
```

Se invoca

```
cuenta := CuentaBancaria new
 inicializarConNumero: 1234 conTitular:'Juan'
```

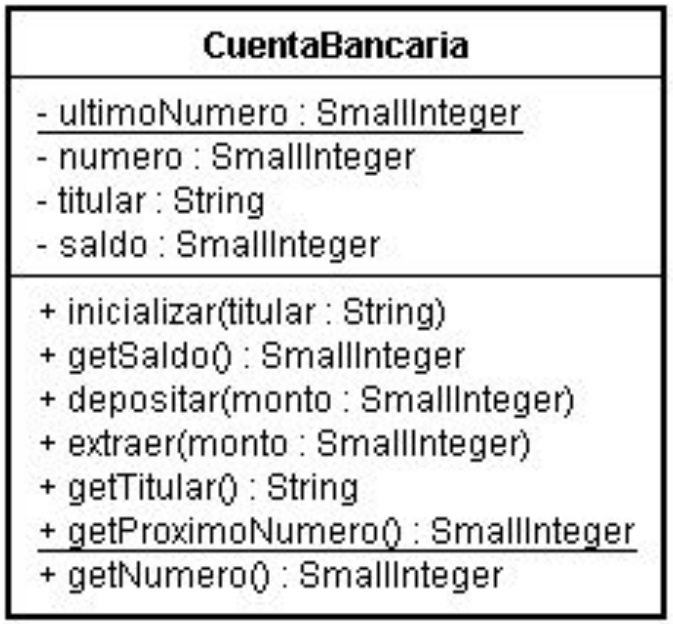
¿Y si el número de cuenta tuviera que ser incremental?

Solución:

- Agregar un atributo *ultimoNumero* que mantenga un único valor para la clase y todas sus instancias

Atributos de clase

- En Smalltalk se declaran en “classVariableNames”
- En Java se declaran *static*



A modo de repaso: implementación

Escribimos prueba

```
PruebaCuentaBancaria >> testAutoIncremental
    cuenta1 := CuentaBancaria new
            inicializarConTitular: 'Juan Pérez'.
    cuenta2 := CuentaBancaria new
            inicializarConTitular: 'Ana García'.
    assert: ( (cuenta2 getNumero) – (cuenta1 getNumero) = 1)
```

Nos aseguramos de que no pase

Implementamos la solución

Nos aseguramos de que corra

Mensajes enviados a la clase

¿Qué hicimos cuando escribimos … ?

```
lista := CuentaCorriente new.
```

CuentaCorriente es una clase

Métodos de clase

- En Smalltalk, las clases son objetos
- En Java, se declaran *static*

Inicialización en Smalltalk (3)

Con método de clase:

```
CuentaBancaria >>
    inicializarConNumero: numero conTitular: titular
```

Se invoca

```
cuenta := CuentaBancaria
 inicializarConNumero: 1234 conTitular:'Juan'
```

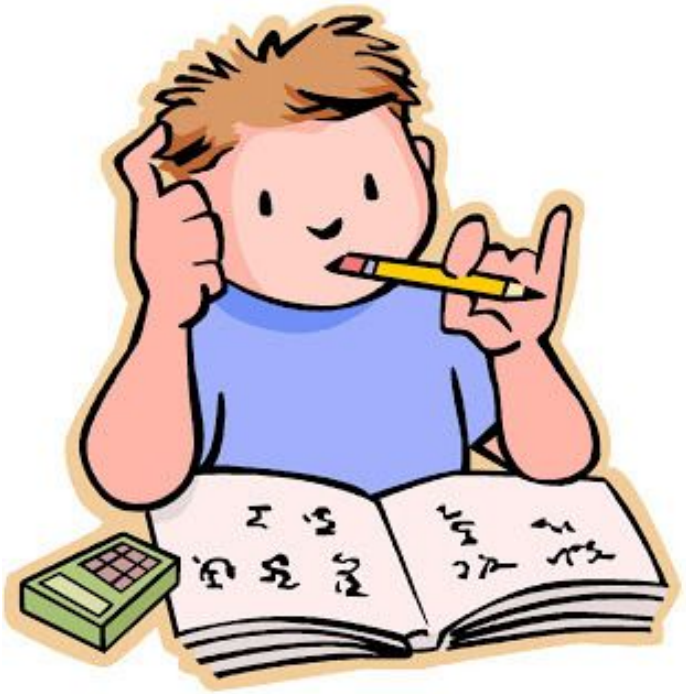
Recapitulación

Recapitulación: preguntas

¿Para qué sirve un atributo de clase?

¿Y un método de clase?

¿Cómo inicializamos objetos en Smalltalk?



Más sobre excepciones

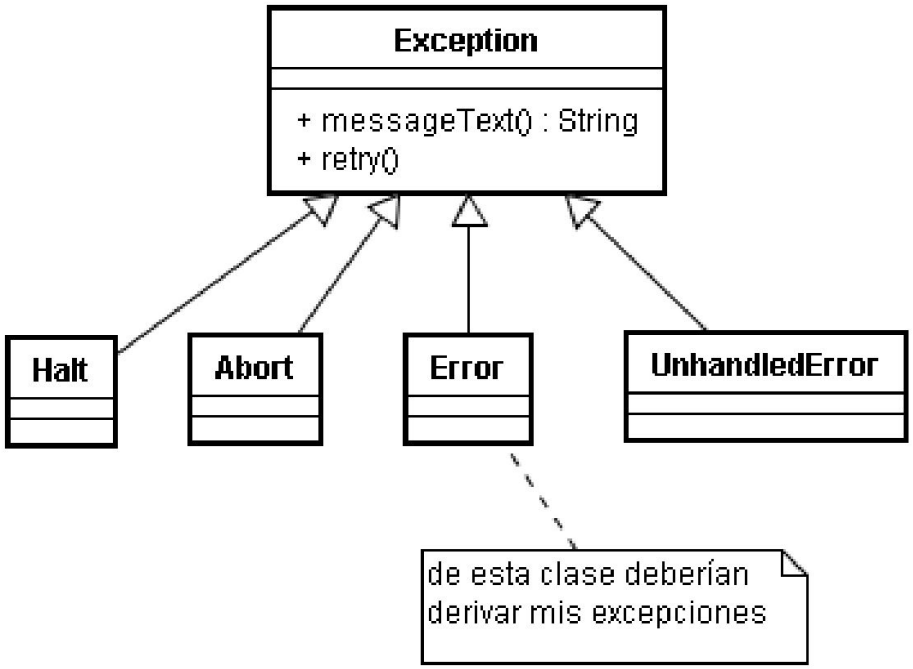
Cuándo lanzar excepciones

- Si en el contexto en el que estamos no hay suficiente información para resolver el potencial problema
  - Hablamos de excepciones cuando el problema no se puede resolver en un determinado contexto
    - Y la lanzamos a un contexto de nivel superior para que resuelva qué hacer
- En el modelo contractual, una excepción ocurre cuando no se cumple una precondición
- También para aislar el código que se usa para tratar problemas del código básico (camino feliz)
- Para crear software más robusto

Qué hacer al capturar

- Resolverla mediante
  - Finalización súbita
  - Continuación ignorando las fallas
  - Avance y recuperación
  - Nuevo intento
- No resolverla y enviarla al contexto invocante
  - La misma
  - Otra excepción

Smalltalk: jerarquía de excepciones



- La jerarquía influye en la captura
  - Cuando decimos capturar un tipo de excepción, capturamos cualquier instancia de esa clase o una descendiente

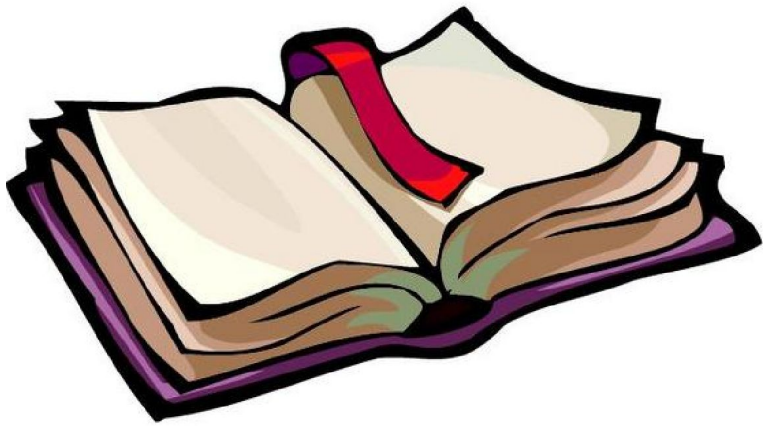
Jerarquías de excepciones propias

Sirven para dar mayor información sobre el tipo de problema

Podrían agregar atributos y métodos

- Pero no es lo más habitual

Cuidar bien la jerarquía

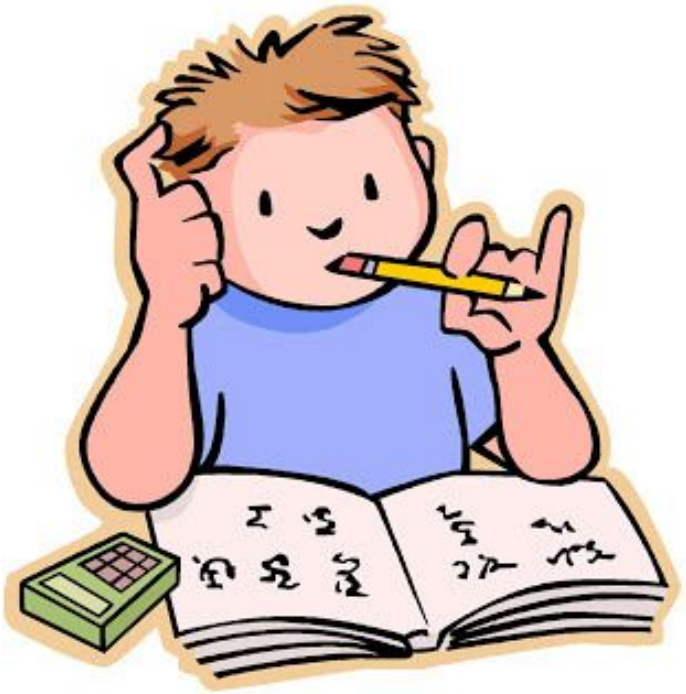


Recapitulación

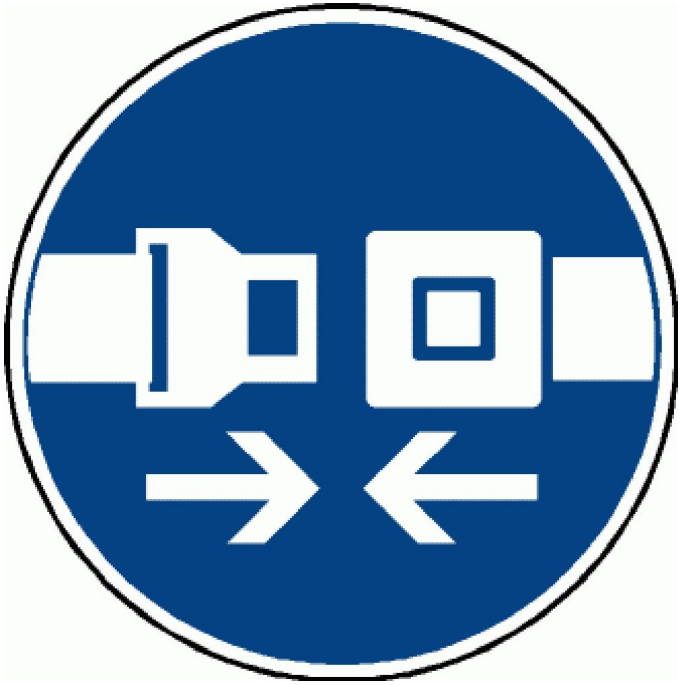
Recapitulación: preguntas

¿Por qué deberíamos crear nuestras clases de excepción?

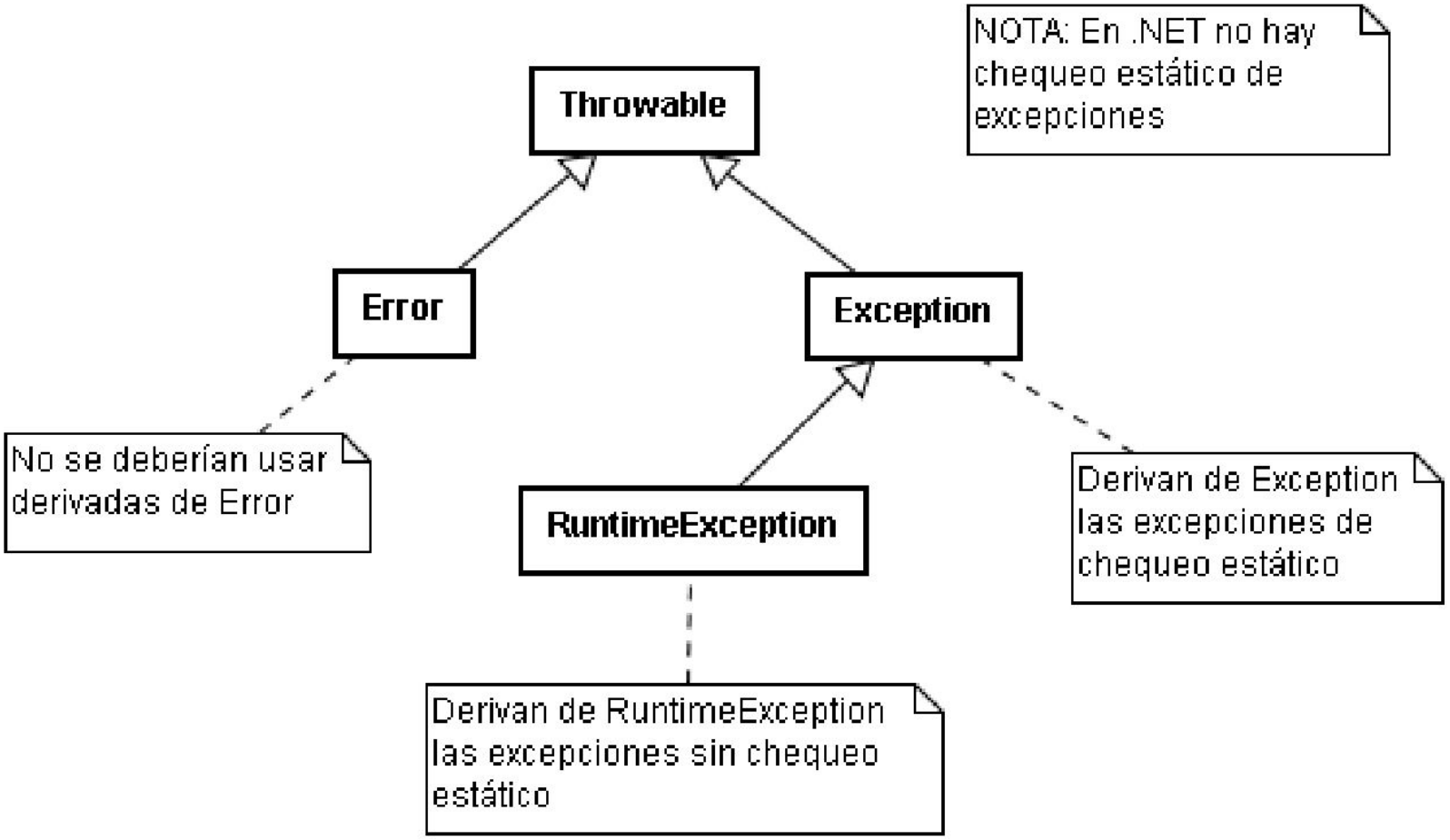
¿En qué casos volvemos a lanzar una excepción al capturar?



Excepciones chequedas en forma estática (Java)



Jerarquía de excepciones (Java)



Excepciones chequeadas

Cláusula “throws” obligatoria

```
public Fraccion dividir (Fraccion y) throws FraccionInvalidaException {
    if (y.numerador == 0)
        throw new FraccionInvalidaException ( );
    int numerador = this.numerador * y.denominador;
    int denominador = this.denominador * y.numerador;
    return new Fraccion(numerador, denominador);
}
```

- A lo sumo se puede declarar un ancestro

En redefiniciones, mantener y no agregar

- Para mantener el polimorfismo: muy molesto

Obligación de capturar: chequeada por el compilador

Posible captura (1)

```
public Fraccion divisionMultiple ( Fraccion [ ] x, Fraccion [ ] y ) {
    Fraccion suma = new Fraccion (0, 1);
    try {
        for (int i = 0; i < 10; i++) {
            Fraccion d = x[i].dividir ( y [i] );
            suma = suma.sumar(d);
        }
    }   catch (FraccionInvalidaException e) {
            tratamientoExcepcion(e);
    }
    return suma;
}
```

Posible captura (2)

```
public Fraccion divisionMultiple ( Fraccion [ ] x, Fraccion [ ] y )
                        throws FraccionInvalidaException {
    Fraccion suma = new Fraccion (0, 1);
        for (int i = 0; i < 10; i++) {
            Fraccion d = x[i].dividir( y[i] );
            suma = suma.Sumar(d);
        }
    return suma;
}
```

Posible captura (3)

```
public Fraccion divisionMultiple ( Fraccion [ ] x, Fraccion [ ] y) {
    Fraccion suma = new Fraccion (0, 1);
    try {
        for (int i = 0; i < 10; i++) {
            Fraccion d = x[i].dividir( y[i] );
            suma = suma.sumar(d);
        }
    }   catch (FraccionInvalidaException e) { }
    return suma;
}
```

Lenguajes y excepciones

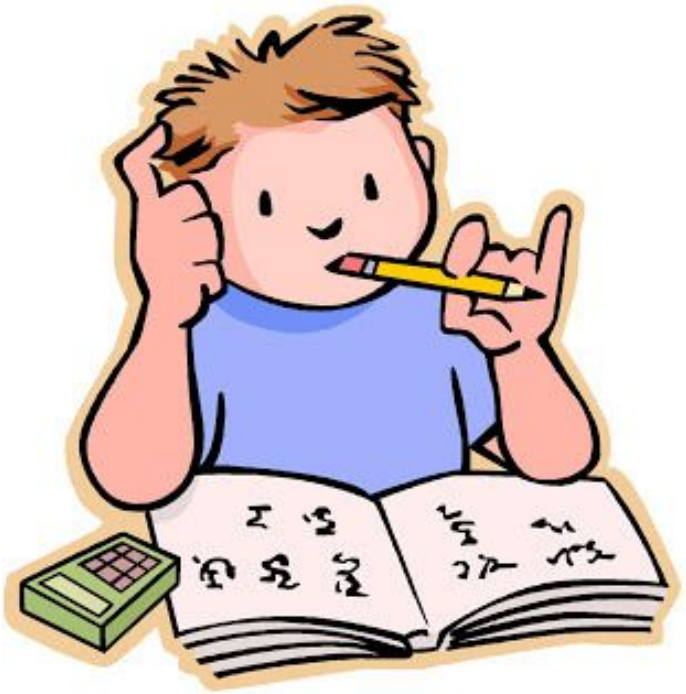
- Excepciones chequeadas
  - Son más seguras
  - Molesta tener que capturarlas sí o sí
  - Limita la redefinición, al no poder agregar nuevas excepciones
    - Se hizo para cumplir con el principio de substitución
- Microsoft diseñó .NET sin excepciones chequeadas
- Ojo: Java permite ambas
  - Aunque es una decisión de diseño

Recapitulación

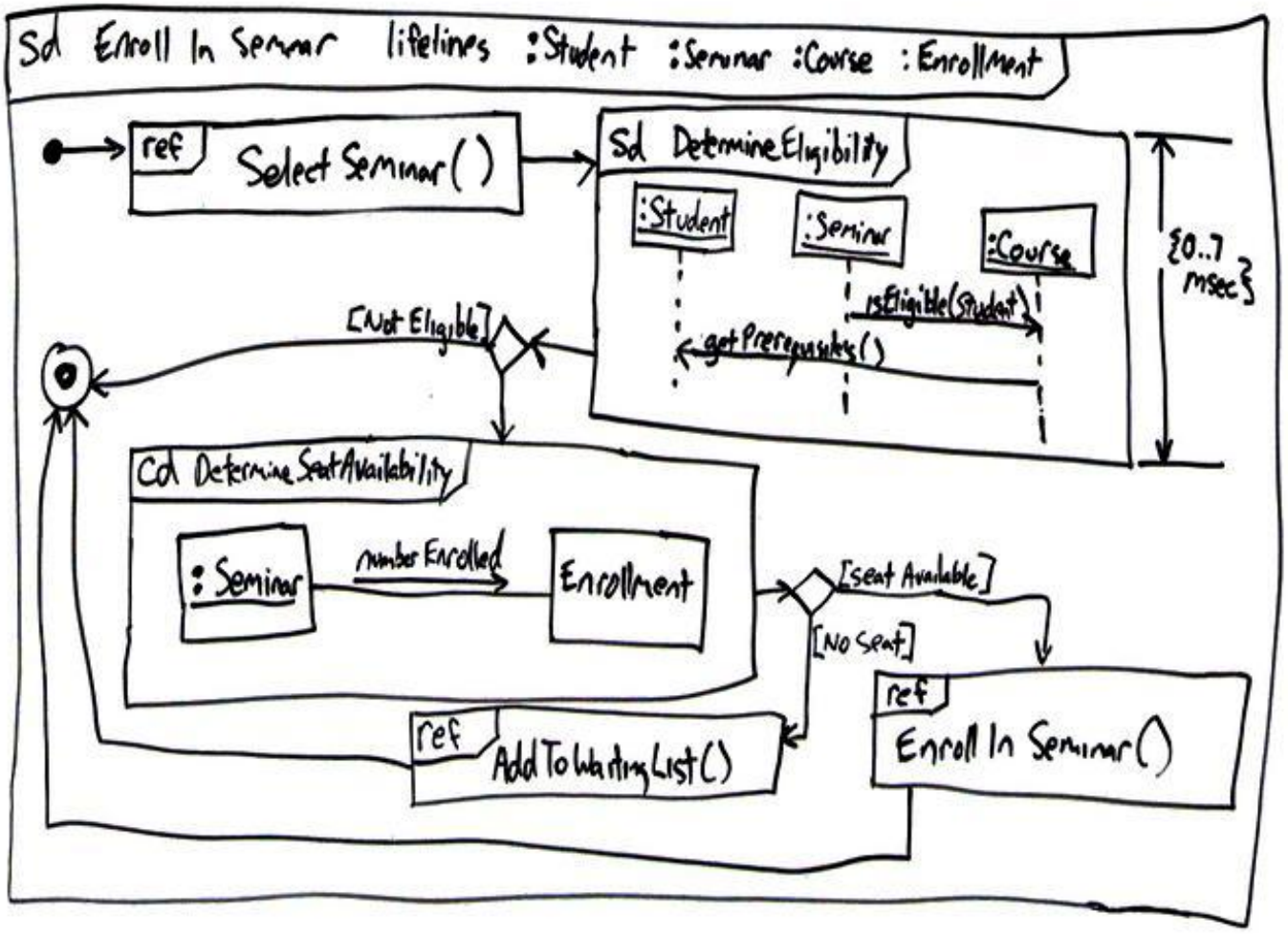
Recapitulación: preguntas

¿Qué significa exactamente que Java tiene un mecanismo de excepciones chequeadas en tiempo de compilación?

¿Qué deberíamos hacer si no deseamos usarlo?



Más UML



fiuba algo3

Diagrama de secuencia

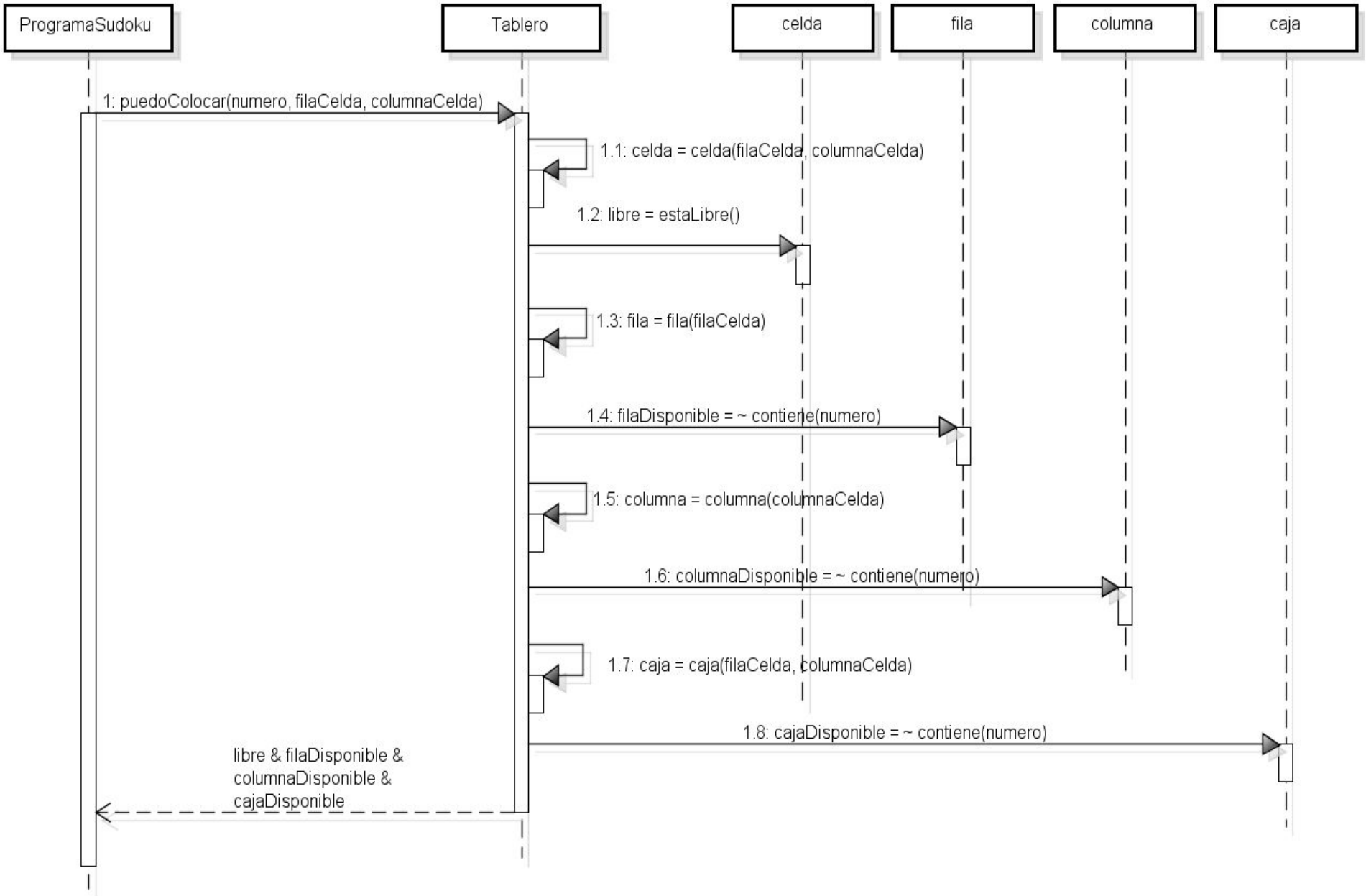
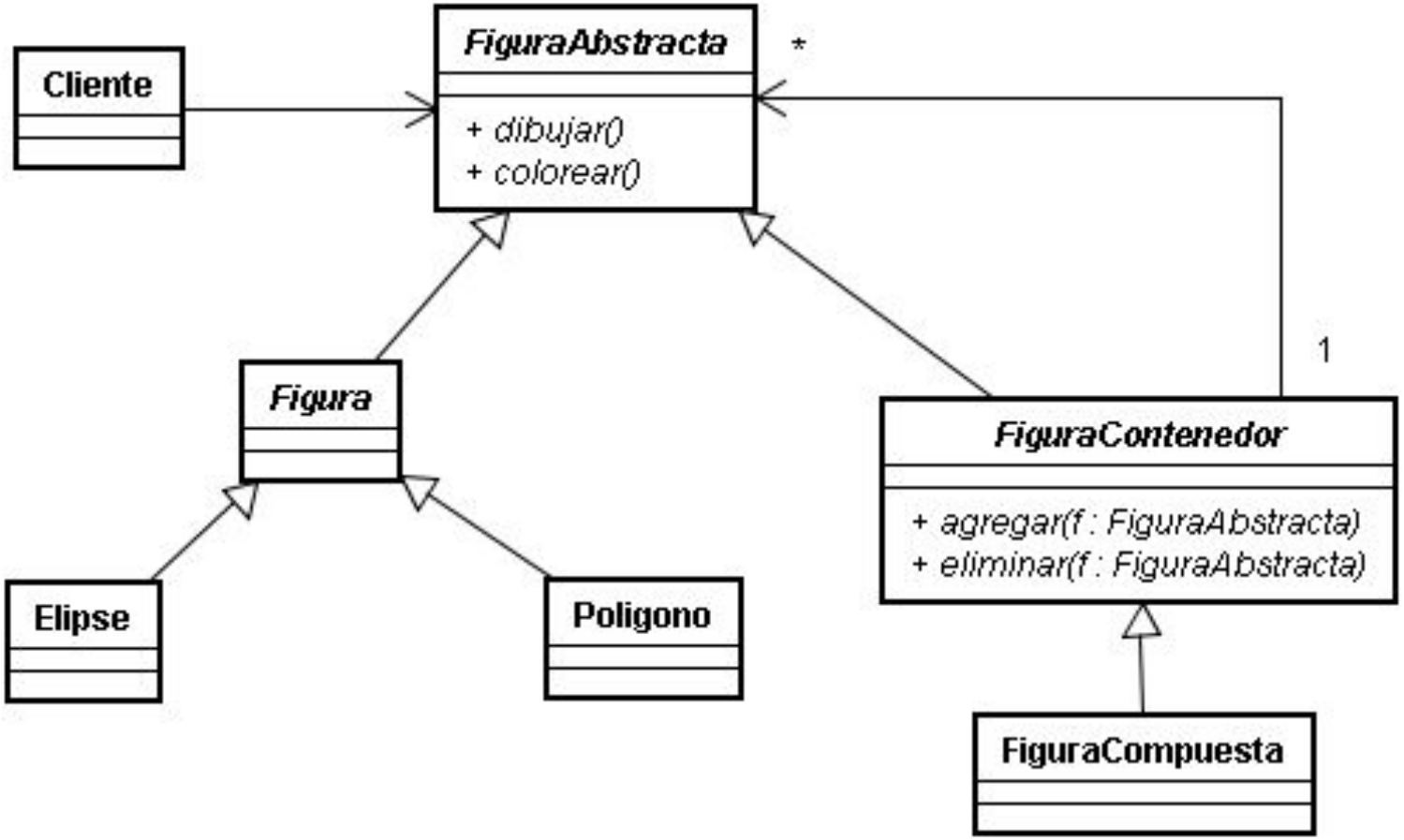


Diagrama de clases

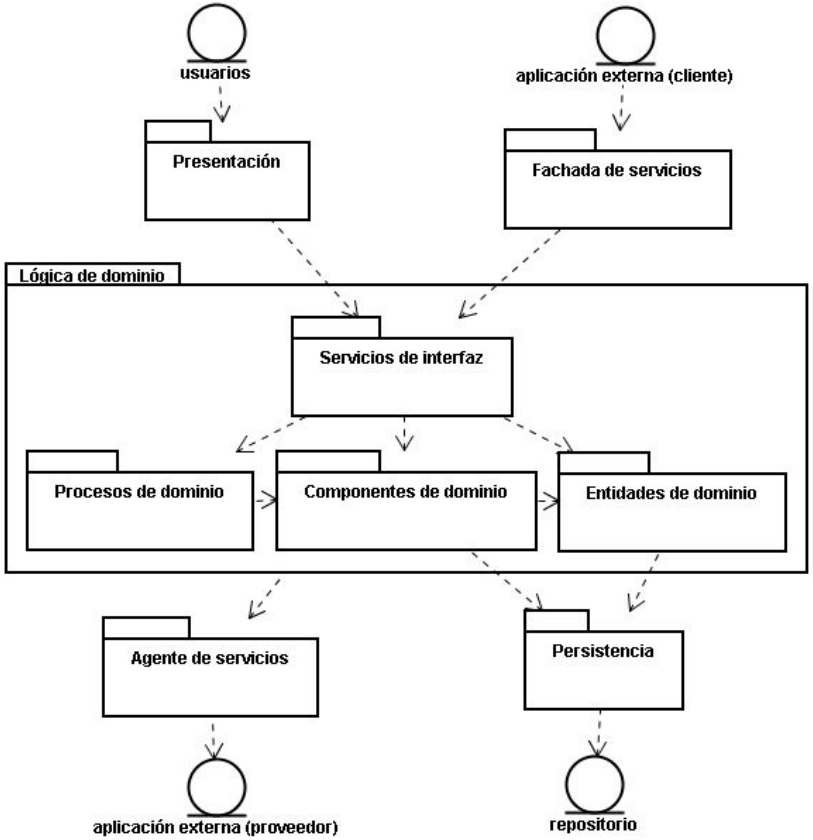


Paquetes

Agrupación de clases

En Java, anidables

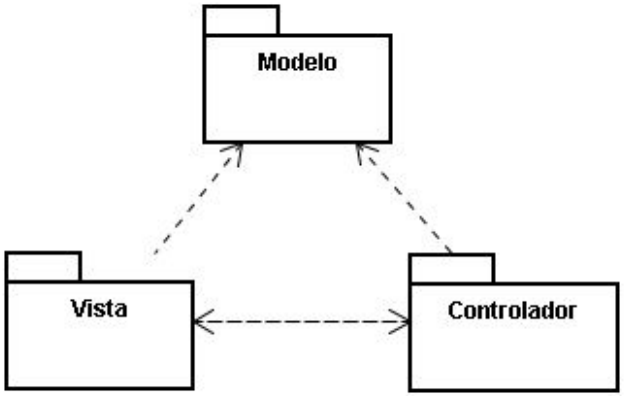
Para manejar la complejidad y modularizar

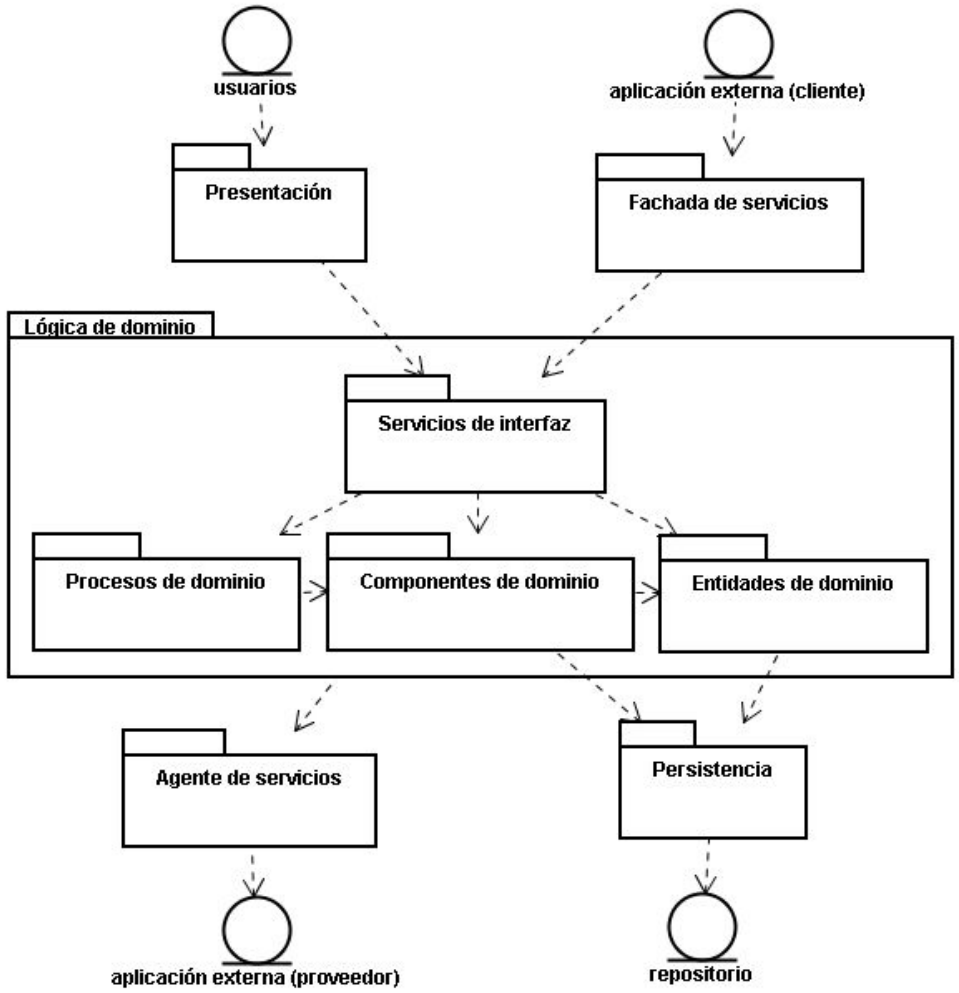


Paquetes en Java

- Toda clase está en un paquete
  - Hay un paquete “default”: mala idea…
  - ArrayList es java.util.ArrayList
  - import java.util.*;
  - import java.util.ArrayList;
- Sirve para resolución de nombres
- Cada clase pública en un archivo fuente separado
  - El paquete se indica en una cláusula “package”
  - package carlosFontela.cuentas;

Diagramas de paquetes





Estados, eventos, transiciones

Estado

- representado por el conjunto de valores adoptados por los atributos de un objeto en un momento dado
- situación de un objeto durante la cual satisface una condición, realiza una actividad o espera un evento

Evento

- Estímulo que puede disparar una transición de estados
- Especificación de un acontecimiento significativo
- Señal recibida, cambio de estado o paso de tiempo
- Síncrono o asíncrono

Diagrama de estados UML: ajedrez

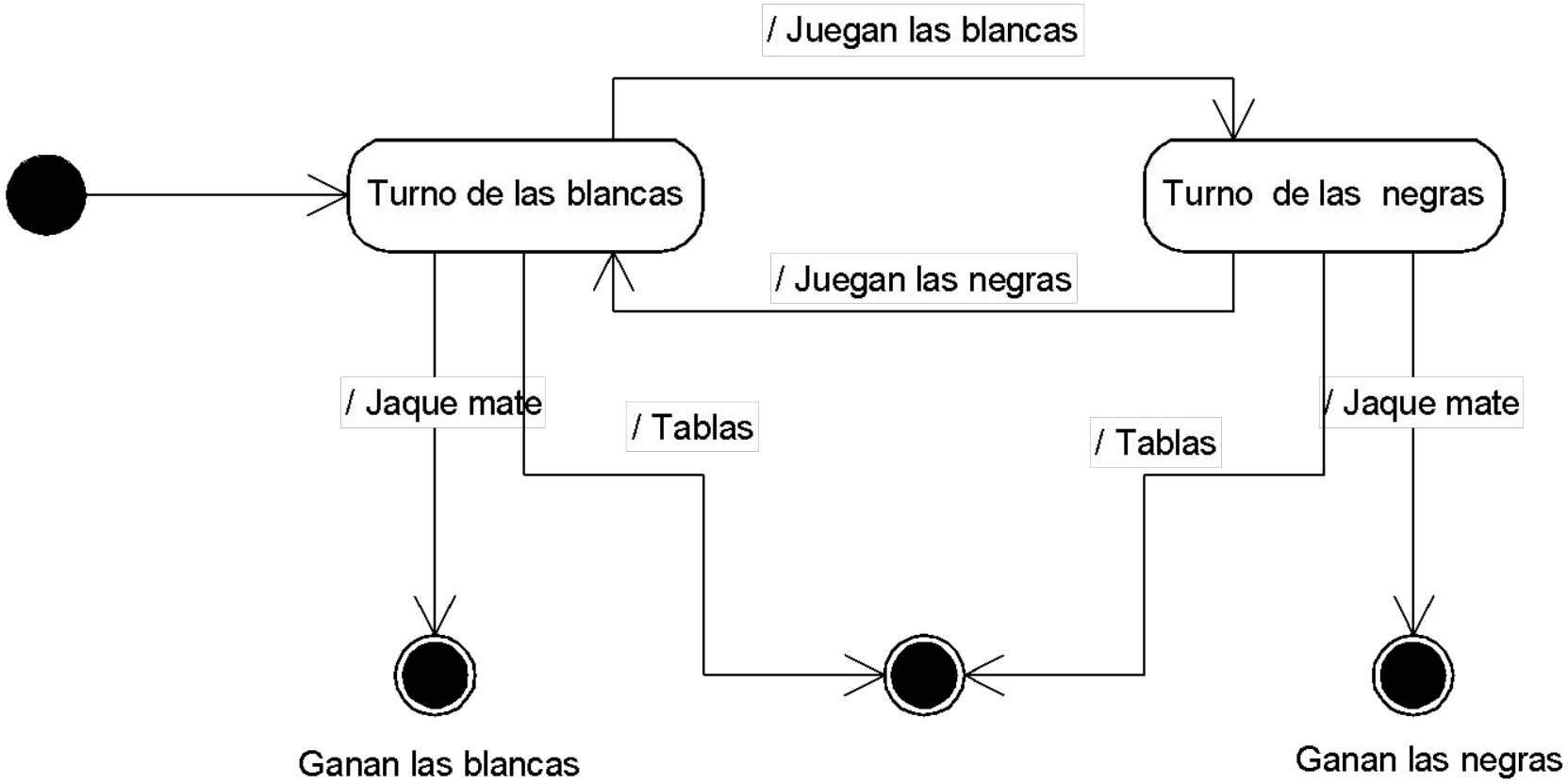


Diagrama de estados UML: estados civiles (1)

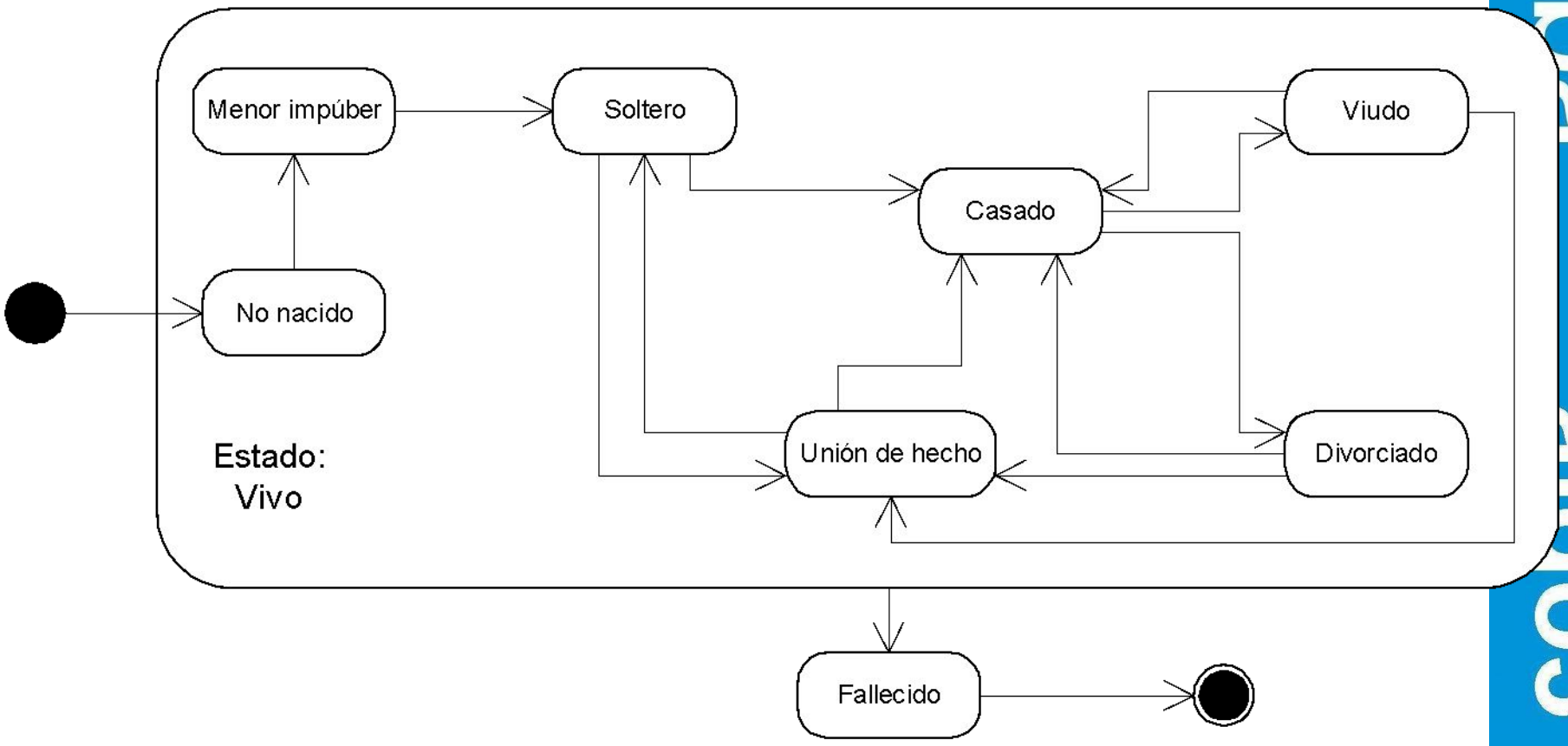
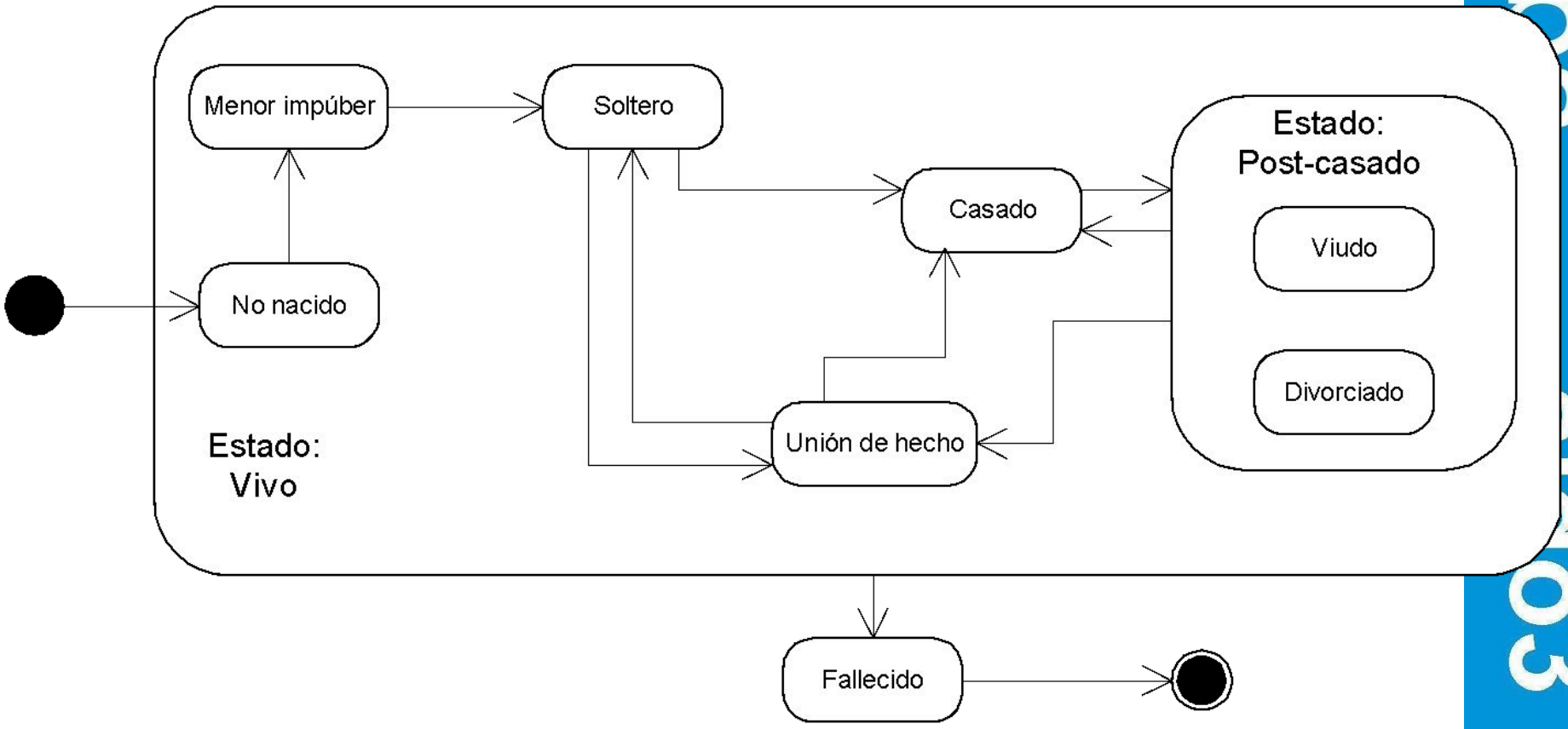


Diagrama de estados UML: estados civiles (2)



UML

Usos de UML

Para discutir diseños antes y durante la construcción

Para generar documentos que sirvan después de la construcción

Destinatarios humanos

- Es lo más habitual

Generación de software

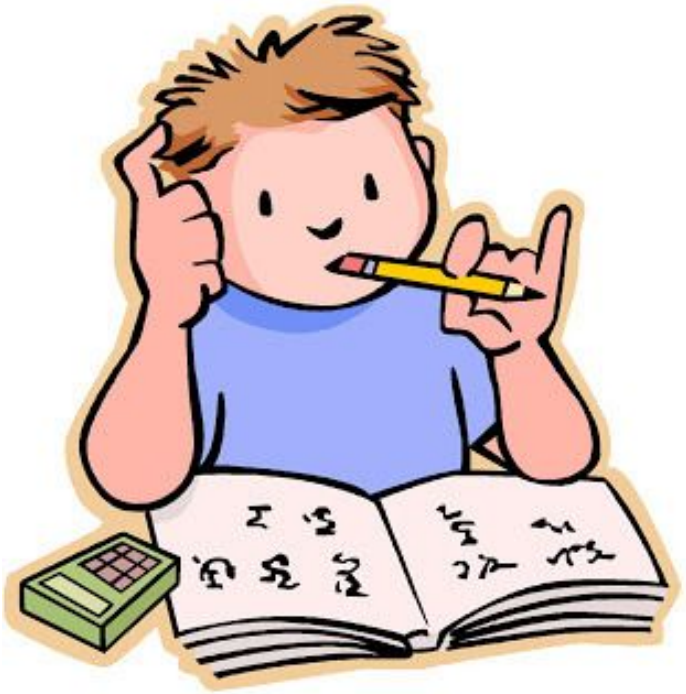
- Model Driven Development
- No es el foco en Algoritmos III

Recapitulación

Recapitulación: preguntas

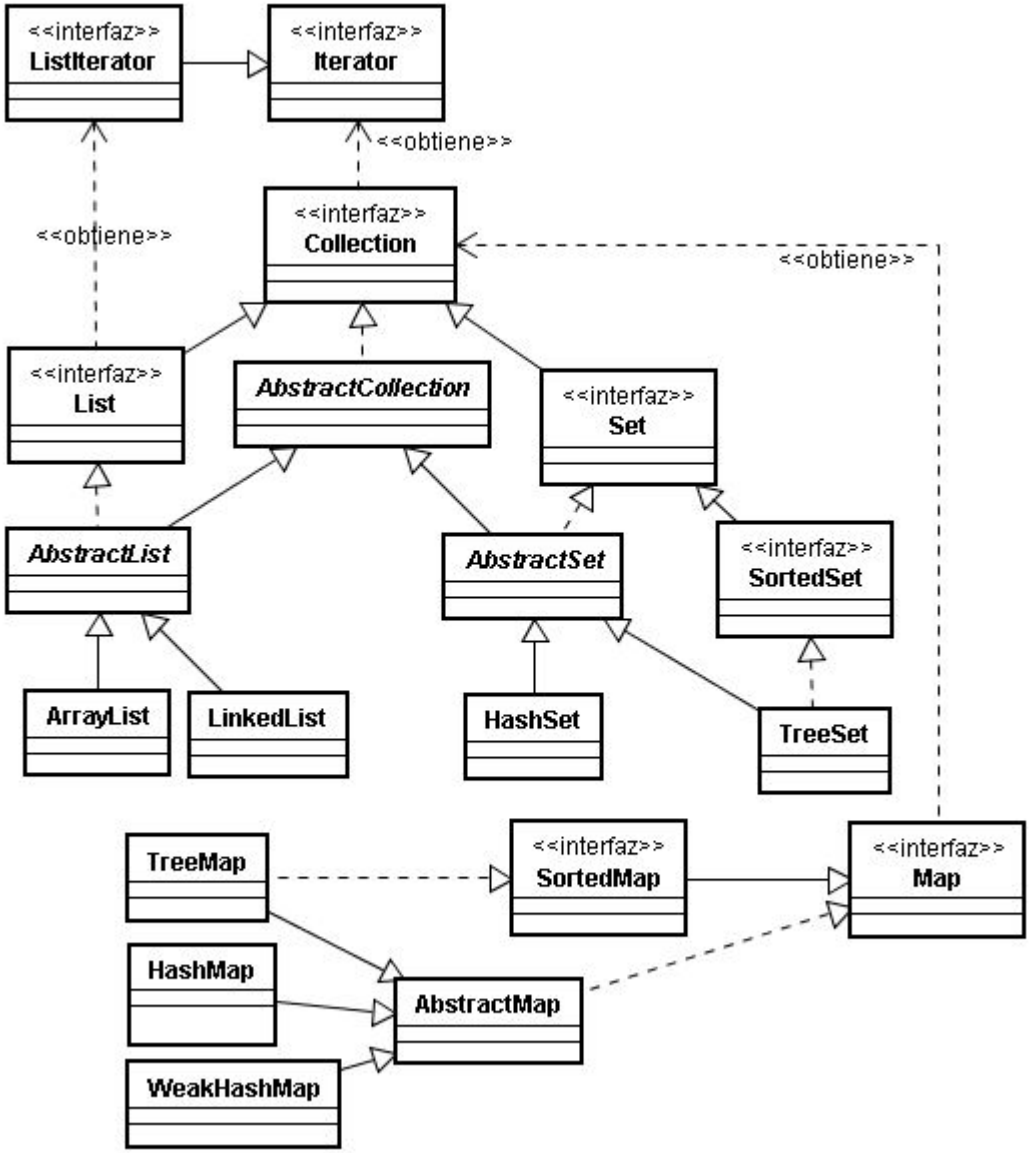
¿Cuándo usarían diagramas de estados?

¿Por qué enseñamos / exigimos UML en un curso de programación?



Colecciones, iteradores y genericidad (Java)

Colecciones de java.util 1.4



fiuba algo3

Java 1.4 ≈ Smalltalk (en las colecciones)

Elementos de tipo Object => sirven para cualquier tipo de datos

```
unaLista.add(new Cuenta());
unaLista.add(new Elipse());
```

¡Pero no es Smalltalk!

No admiten tipos primitivos

```
unaLista.add(4);// no funciona en Java 1.4
```

“Cualquier tipo” al recuperar: ¿?

```
Cuenta c = unaLista.get(pos);
    // error: Cuenta no es Object
Cuenta c = (Cuenta) (unaLista.get(pos));
```

Iteradores: definición y uso

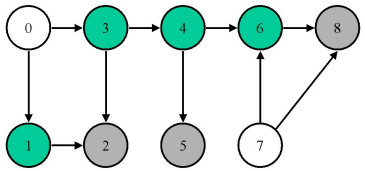
Objetos que saben cómo recorrer una colección, sin ser parte de ella

Interfaz:

- Tomar el primer elemento
- Tomar el elemento siguiente.
- Chequear si se termina la colección

Un ejemplo:

```
List vector = new ArrayList();
for(int j = 0; j < 10; j++) vector.add(j);
Iterator i = vector.iterator();      // pido un iterador para vector
while ( i.hasNext() )                // recorro la colección
        System.out.println( i.next() );
```



Iteradores y colecciones

Toda clase que implemente Collection puede generar un Iterator con el método iterator

Nótese que Iterator es una interfaz

Pero está implementada para las colecciones definidas en java.util.

Iteradores: para qué

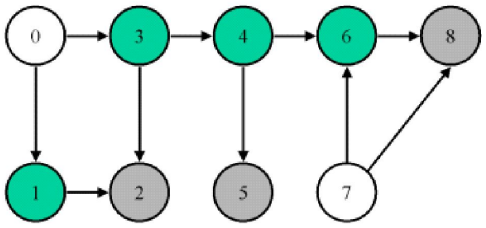
Llevan la abstracción a los recorridos de colecciones

Facilitan cambios de implementación

```
Collection lista = new ArrayList ( );
Iterator i = lista.iterator();   // pido un iterador para lista
while ( i.hasNext() )     // recorro la colección
        System.out.println( i.next() );
```

No se necesita trabajar con el número de elementos

Convierten a las colecciones en simples secuencias



Iteradores en Smalltalk

Modelo más simple

Analizar

```
celdas do: [:celda |  (celda contiene: numero)
  ifTrue: [ encontrado := true ] ].
```

celdas referencia una instancia de

OrderedCollection

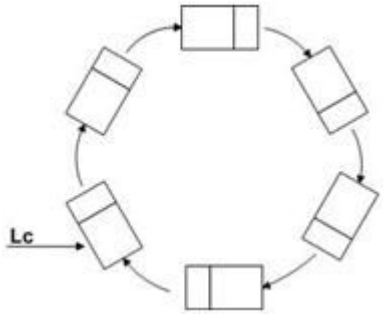
Ejercicio: lista circular (1)

¿Qué es una lista circular?

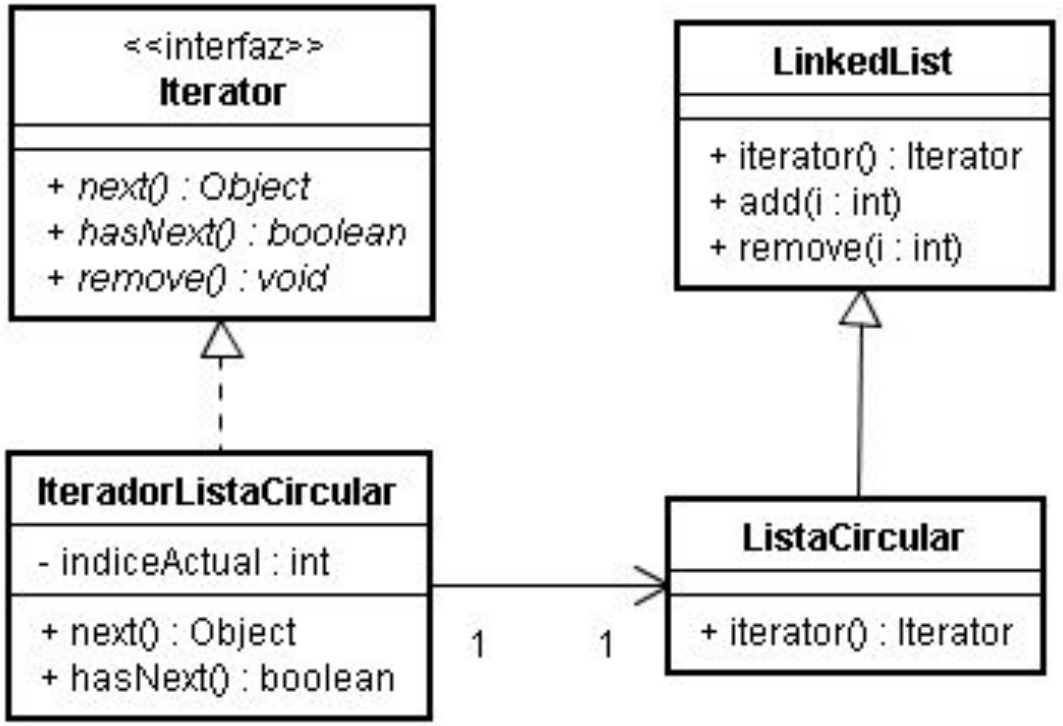
- Definición: una lista que se recorre indefinidamente, de modo tal que al último elemento le sigue el primero

Es un caso particular de LinkedList

- ¿Qué cambia?
- ¿Nada?
- ¿Sólo la forma de recorrerla?
- => El iterador es diferente



Ejercicio: lista circular (2)



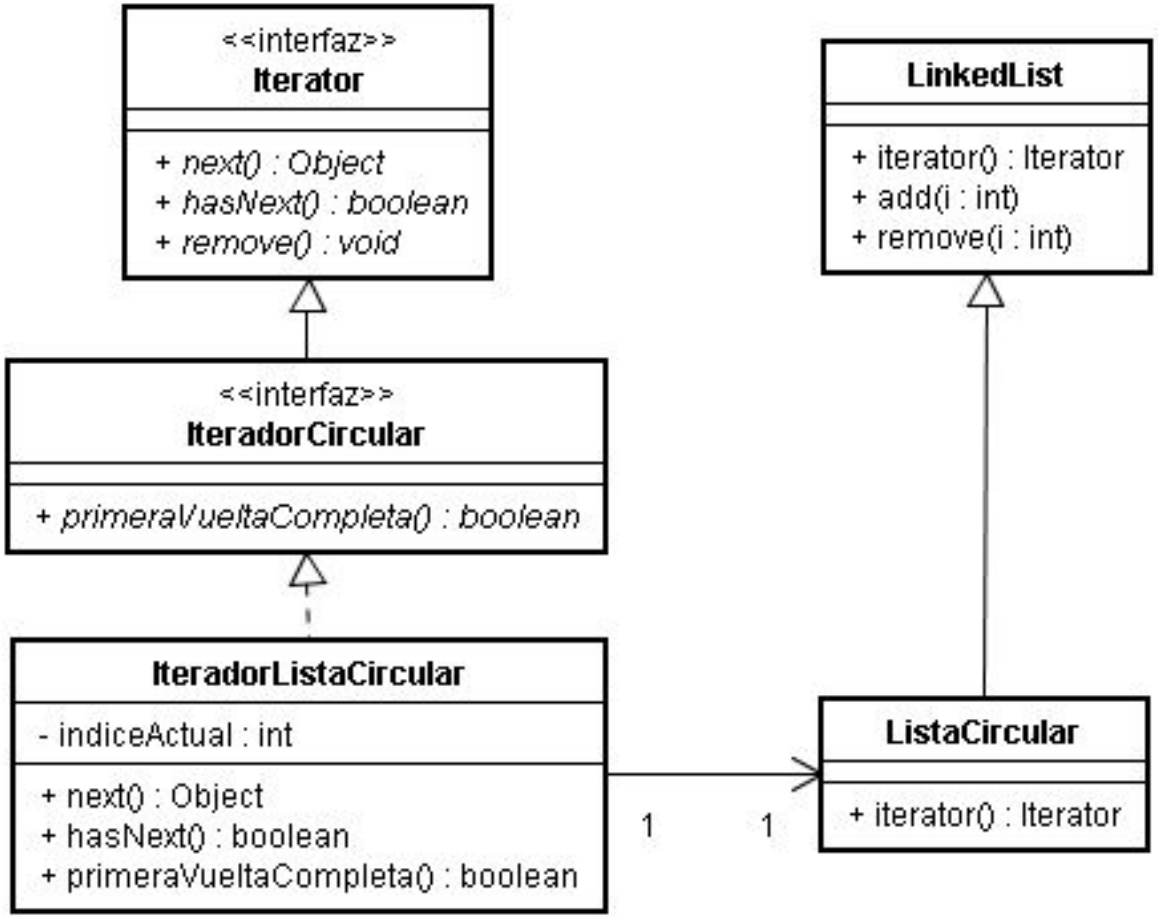
Ejercicio: lista circular (3)

```
public class ListaCircular extends LinkedList {
        public Iterator iterator( ) {
            return new IteradorListaCircular(this);
    }
}
```

Implementar la clase IteradorListaCircular

Con sus métodos next() y hasNext()

Ejercicio lista circular: otra visión



Genericidad: los tipos también son parámetros

Sin

```
List v = new ArrayList( );
String s1 = “Una cadena”;


// s1 pasa como Object:
v.add(s1);


// get obtiene un Object
// “puenteo” el chequeo:
String s2 =
  (String)v.get(0);
```

Con

```
List<String> v =
    new ArrayList<String>(
  );
String s1 = “Una cadena”;

// el compilador verifica
  que s1 sea un String:
v.add(s1);

String s2 = v.get(0);
```

Genericidad (más allá)

En métodos, el compilador infiere el tipo genérico:

```
public static <T> void eliminarElemento (List<T> lista, int i) { … }
eliminarElemento (listaConcreta, 6);
```

Mejoras:

- Robustez en tiempo de compilación
- Legibilidad

Cuestiones avanzadas

```
public static <T extends Comparable > void ordenar (T[ ] v) { … }
public static <T > copy (List<T> destino, List<? extends T> origen) { … }
public static <T, S extends T> copy (List<T> destino, List<S> origen) { … }
```

Genericidad: Java vs. .NET

Java

usa la genericidad sólo para tiempo de compilación

- No llega al bytecode => compatibilidad hacia atrás
- No hay información del tipo completa en tiempo de ejecución

.NET

mantiene la información de tipos completa hasta tiempo de ejecución

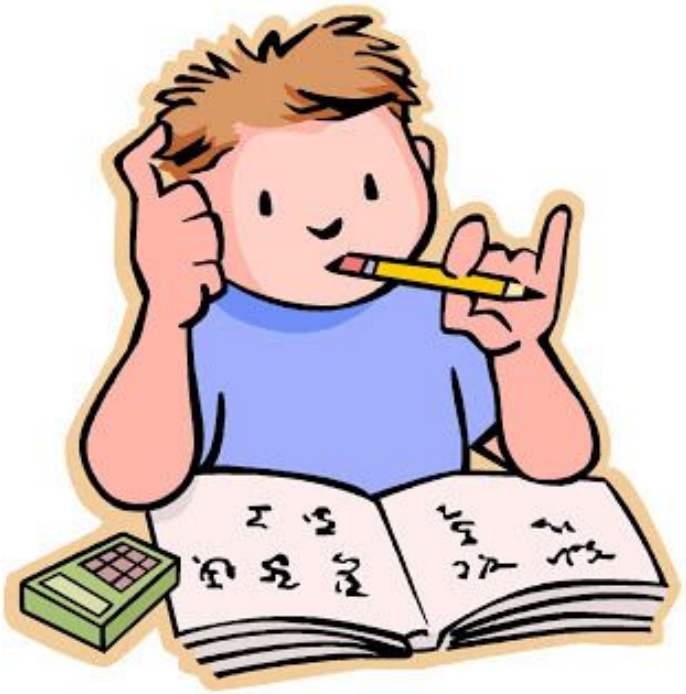
- Pero generó una biblioteca de clases nueva => sin compatibilidad hacia atrás

Recapitulación

Recapitulación: preguntas

¿Qué ventajas aporta la genericidad cuando trabajamos con colecciones?

¿Para qué se usan las interfaces?



Claves

- Inicialización debería dejar al objeto en un estado válido
- Excepciones cuando no hay suficiente información en el contexto del problema
- UML es una herramienta de modelado
  - Para discutir diseños antes del código
  - Para generar documentos que sirvan después de la construcción
- Genericidad lleva la idea del tipeo estático a las colecciones
- También permite que los tipos sean parámetros

Lecturas obligatorias

"What's a Model For?", Martin Fowler

http://martinfowler.com/distributedComputing/purpose.pdf

Lecturas optativas

UML Distilled 3rd Edition, Martin Fowler, capítulo 1 “Introduction”

- Hay edición castellana de la segunda edición
- Debería estar en biblioteca

UML para programadores Java, Robert Martin, capítulo 2 “Trabajar con diagramas”

- No está en la Web ni en la biblioteca

Orientación a objetos, diseño y programación, Carlos Fontela 2008, capítulo 9: “Excepciones”

Qué sigue

Repaso y revisión lecturas obligatorias

Calidad de código

Primer parcial 8/10