FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Excepciones Colecciones e iteradores

Carlos Fontela
cfontela@fi.uba.ar

A3F

Temario

Excepciones

- Desacoplar código de tratamiento de problemas
- Tratamiento de problemas con un enfoque optimista

Colecciones e iteradores

- En Java 1.4 y .NET 1.1

Estados y diagramas de estados

Manejo de problemas (v1)

```
public Fraccion (int numerador, int denominador) {
        if (denominador == 0)
                throw new ArgumentException( );
        this.numerador = numerador;
        this.denominador = denominador;
}
public Fraccion Dividir (Fraccion y) {
        if (y.numerador == 0)
                throw new DivideByZeroException( );
        int numerador = this.numerador * y.denominador;
        int denominador = this.denominador * y.numerador;
        return new Fraccion(numerador, denominador);
}
```

Manejo de problemas (v2)

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

Clases de excepciones

¿Qué es FraccionInvalidaException?

Una clase de excepción

Debe derivar de Exception o una derivada

```
public class FraccionInvalidaException :
    System.ApplicationException { }
```

Lo que se lanza es un objeto, instancia de esa clase

Manejo de problemas (uso)

```
try {
        Fraccion f2 = new Fraccion (2, 3);
        Fraccion f3 = new Fraccion (0, 3);
        Fraccion f5 = f2.Dividir(f3);
        Console.WriteLine ("Se pudo hacer la división y el resultado es: “
                + f5.ValorReal( ).ToString( ));
}
catch (FraccionInvalidaException e) {
        Console.WriteLine ("No se pudo hacer la división");
}
```

Clases de excepciones

Deben heredar de Exception o una derivada:

- System.Exception en C#
- java.lang.Exception en Java

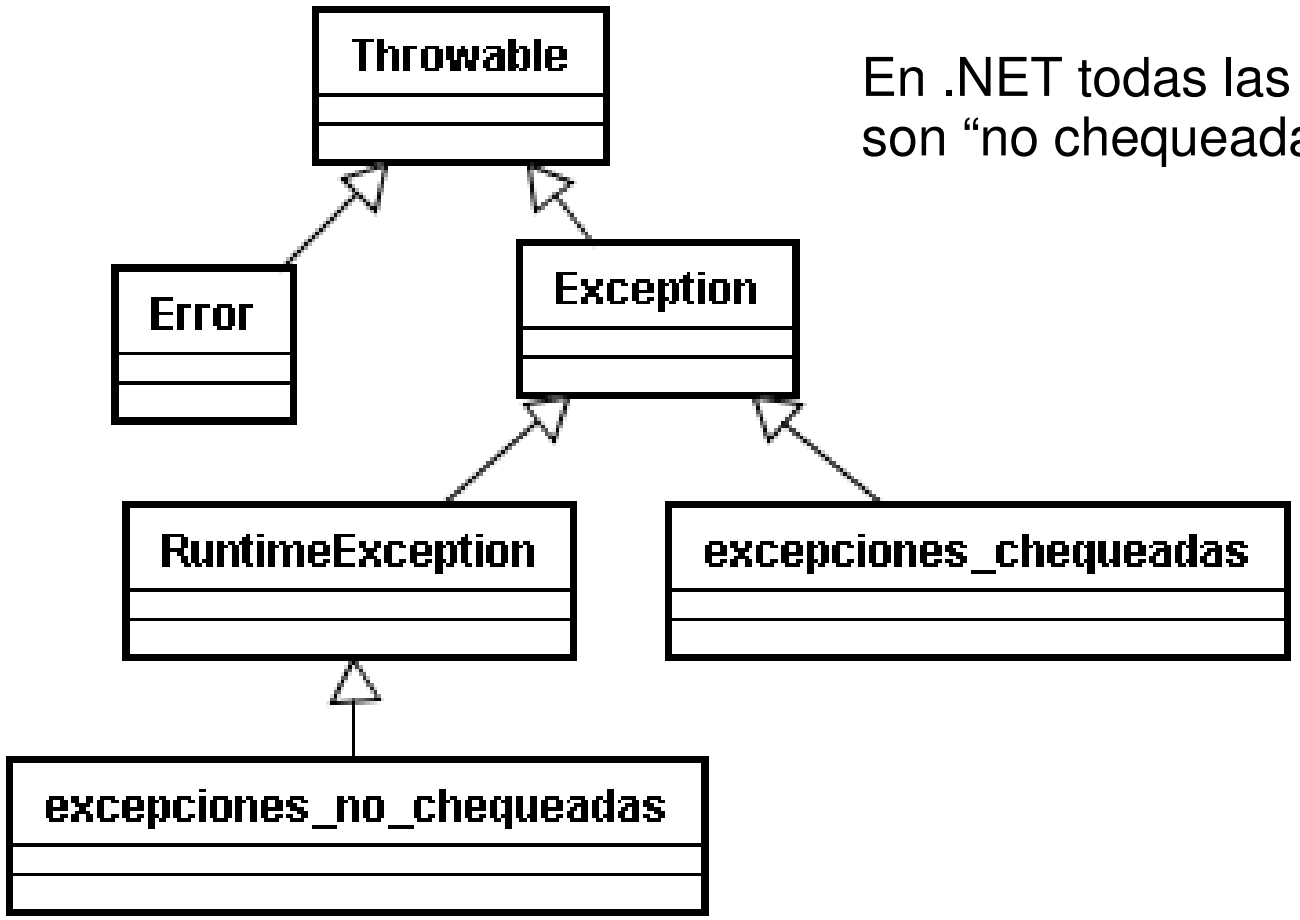
Pueden tener atributos y métodos:

- Para usar en la captura
- En general no se usan

Las jerarquías influyen en la captura:

- Se captura cualquier clase derivada
- Ojo con el orden de captura

Jerarquías de excepciones (Java)



En .NET todas las excepciones son “no chequeadas”

Excepciones chequeadas (Java, 1)

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

Excepciones chequeadas (Java, 2)

Obligación de capturar (I)

```
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) {
            System.err.println("División por cero");
            return new Fraccion (0, 1);
    }
    return s;
}
```

Excepciones chequeadas (Java, 3)

Obligación de capturar (II)

```
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 s;
}
```

Excepciones chequeadas (Java, 4)

Obligación de capturar (III)

```
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 s;
}
```

Lenguajes: enfoques

- Excepciones chequeadas
  - Son más seguras
  - Molesta tener que capturarlas sí o sí
  - Limita la redefinición, al no poder agregar nuevas excepciones
  - Aunque cumple el principio de substitución
- Microsoft diseñó .NET sin excepciones chequeadas
- C++ tiene un enfoque mixto
- Ojo: Java permite ambas
  - Aunque es una decisión de diseño

Comprobación de estados

Enfrentar problemas en tiempo de ejecución:

```
double r = x / y;
```

¡y puede ser 0!

```
Matriz a = b.inversa();
```

¡b puede no ser invertible!

¿Cómo enfrentamos los problemas?

Enfoques conservadores (1)

if (b.invertible())

a = b.inversa();

else System.out.println ("La matriz b no era invertible");

Estoy suponiendo que el resultado no era importante

Y que el mensaje tiene sentido para el usuario final

boolean error = false;

if (b.invertible())

a = b.inversa();

else error = true;

Supongo que el programador cliente sabrá qué hacer con "error"

Enfoques conservadores (2)

```
while ( ! b.invertible() ) {
    System.out.println("La matriz b no es invertible, ingrese otra");
    b.leer();
}
a = b.inversa();
```

Supongo que el usuario final sabe de qué le estoy hablando.

```
do { }
while (! b.invertible() );    // espera que el estado cambie
a = b.inversa();
```

Supongo que otro hilo vendrá en mi ayuda.

Enfoques conservadores: ingenuidad

Los más simples son propios de novatos

Los parámetros de error:

- Envían el problema a un contexto que tal vez no los pueda resolver
- Son tediosos de verificar
- Y en general no se verifican

Enfoques optimistas

El conservador dice:

- “Compruebo primero y luego actúo”

El optimista:

- “Pruebo actuar y resuelvo en el caso de hallar problemas”
- Puede ser mucho más eficiente

¿Cómo encaro el enfoque optimista?

- Excepciones

Colecciones

Agrupan objetos

Se puede operar sobre:

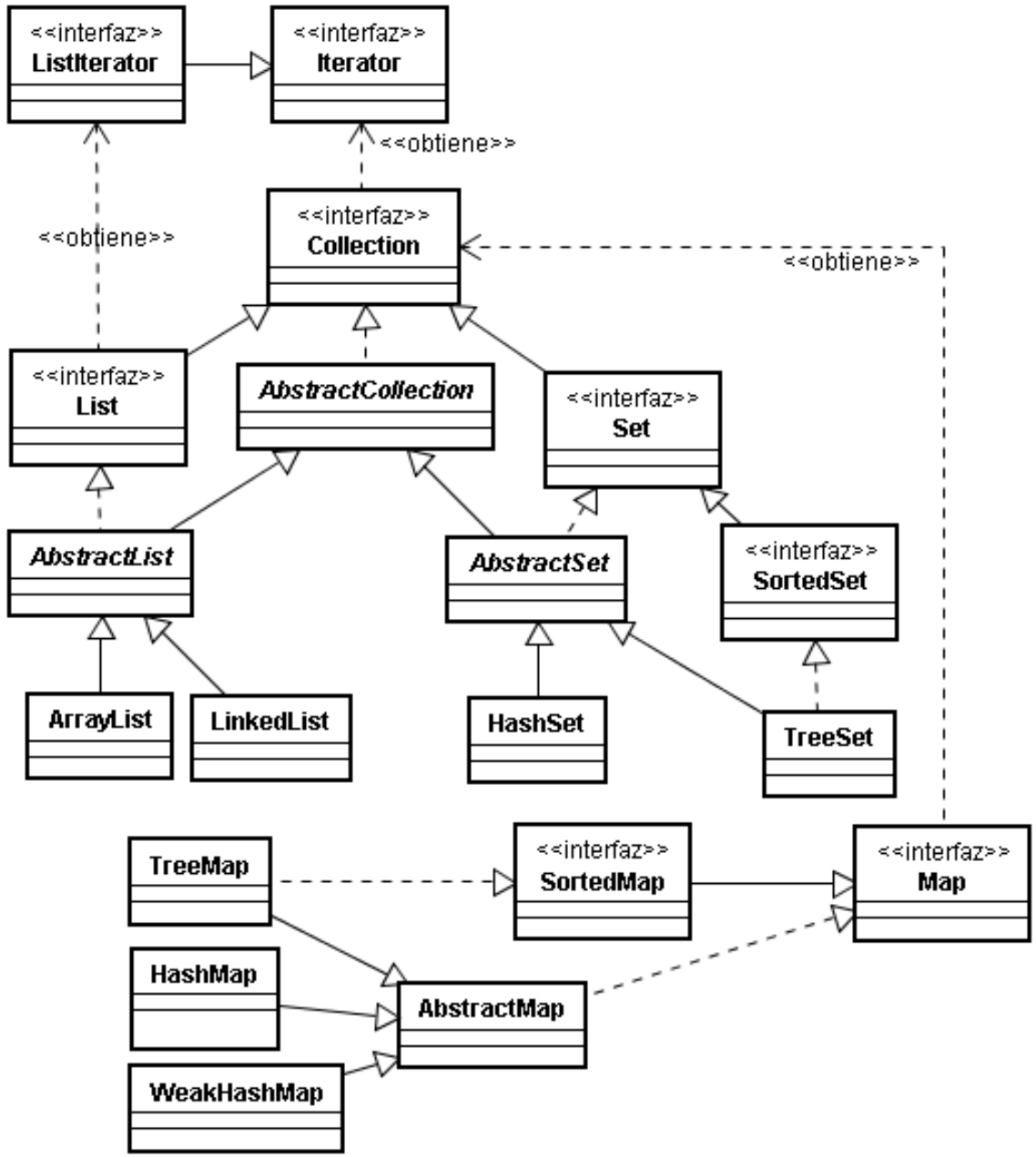
- Un elemento en particular
- Algunos elementos elegidos mediante un filtro
- La colección como conjunto
- Se definen recorridos

Tienen diferentes interfaces, funcionalidades y eficiencias

Arreglos, listas, árboles, etc.

Colecciones de java.util (1)

Las más comunes de Java 1.4:



Colecciones de java.util (2)

Tienen elementos de tipo Object.

- No se sabe qué hay dentro
- “Casteo” para obtener utilidad
- No admiten elementos primitivos.
  - Pero hay clases envolventes: Integer, Boolean, Double, Character, etc.

Colecciones heredadas:

- Vector, Hashtable, Stack, BitSet, Properties, etc.

Clase Collections

Una clase utilitaria de métodos estáticos

Algunos métodos:

void sort (Collection c, Comparator comp)

int binarySearch (Collection c, Object x, Comparator comp)

Object max (Collection c, Comparator comp)

Object min (Collection c, Comparator comp)

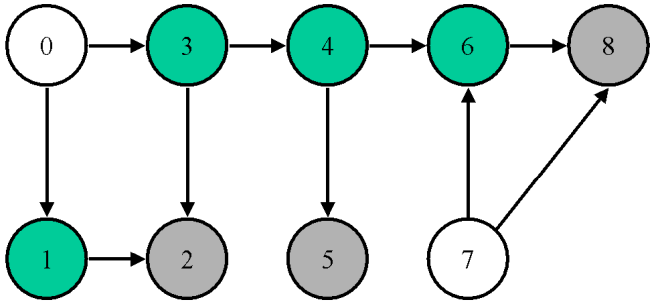
void reverse()

Collection unmodifiableCollection (Collection c)

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.

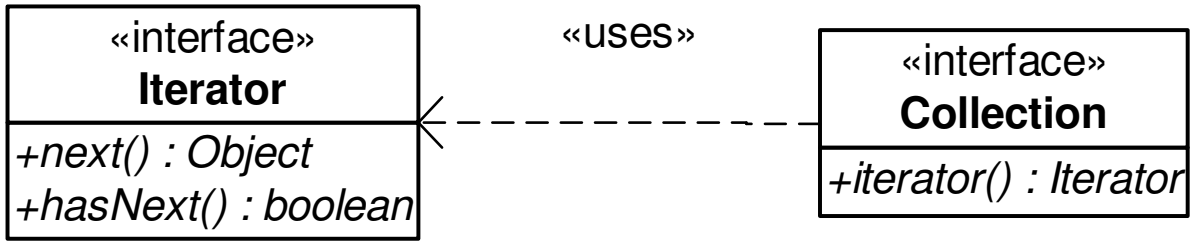
Chequear si se termina la colección

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

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

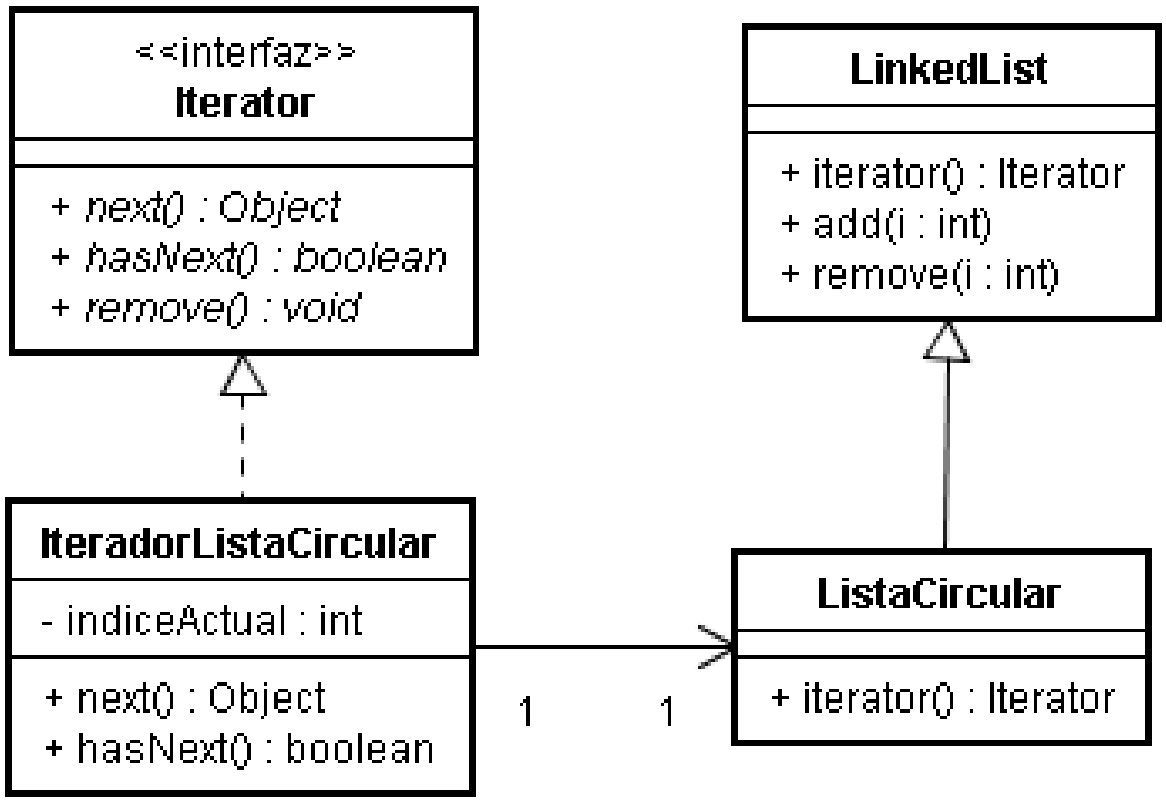
¿Qué cambia?

¿Nada?

¿Sólo la forma de recorrerla?

=> El iterador es diferente

Ejercicio: lista circular (2)



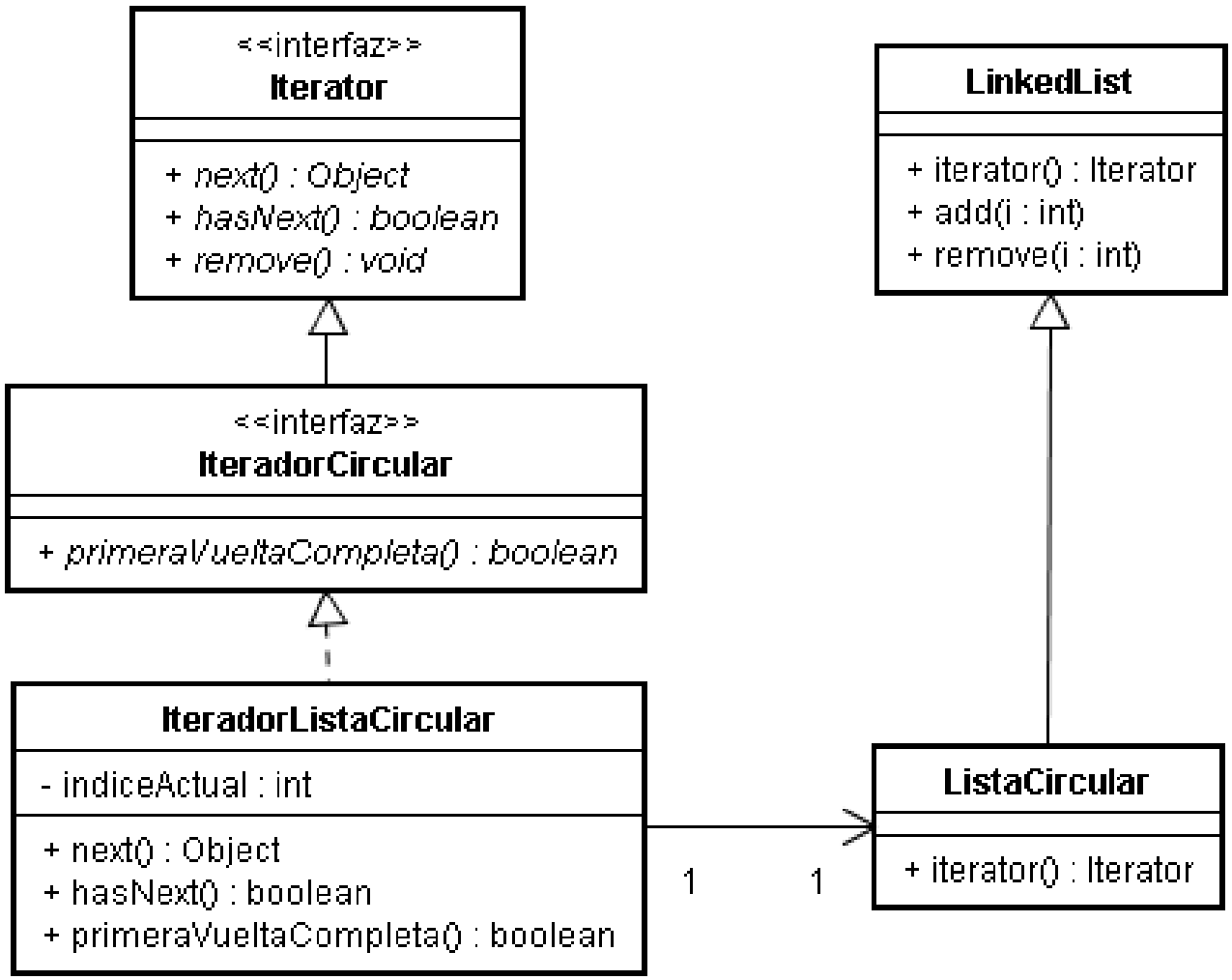
Ejercicio: lista circular (3)

```
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



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

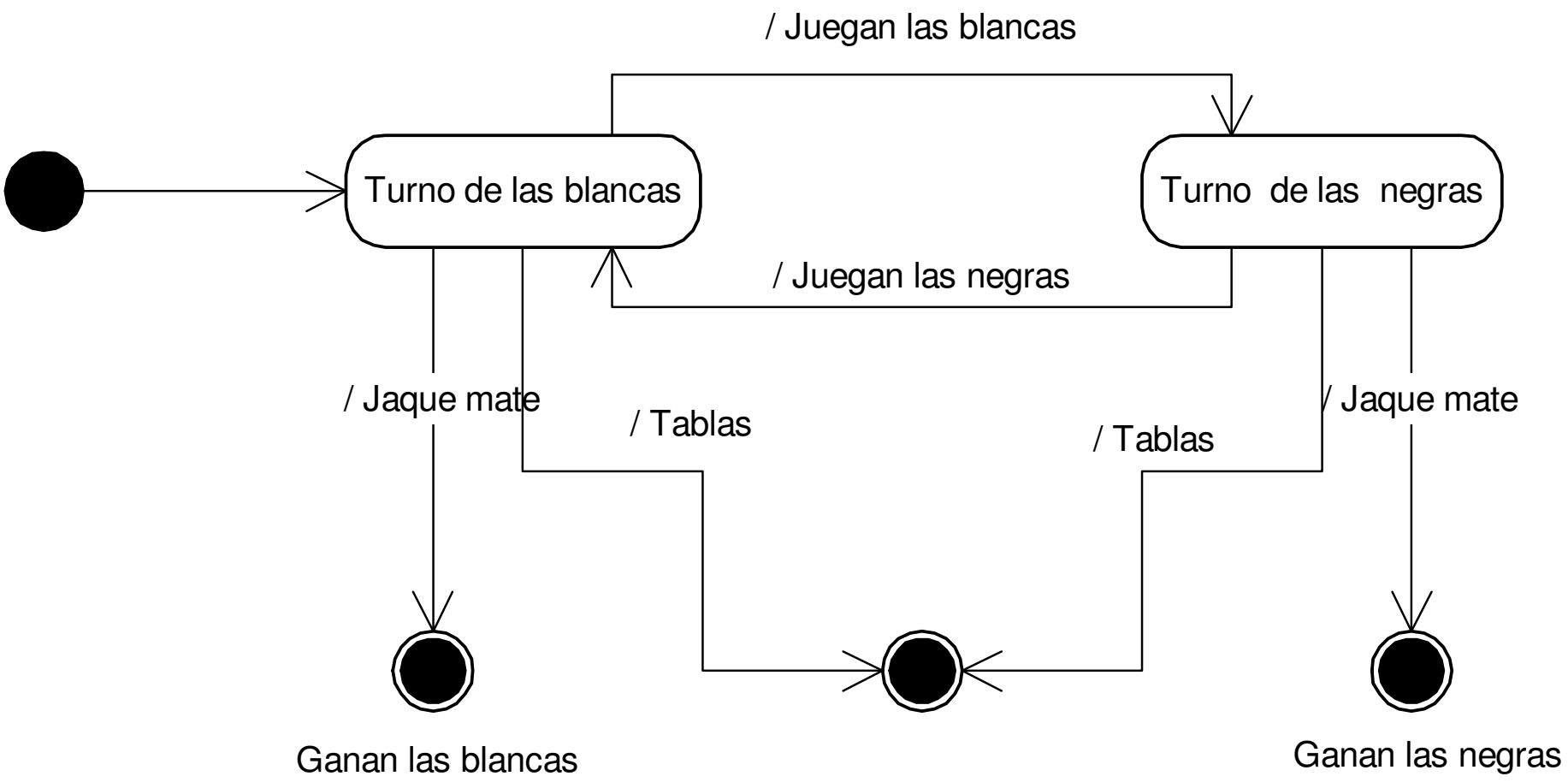


Diagrama de estados UML: estados civiles (1)

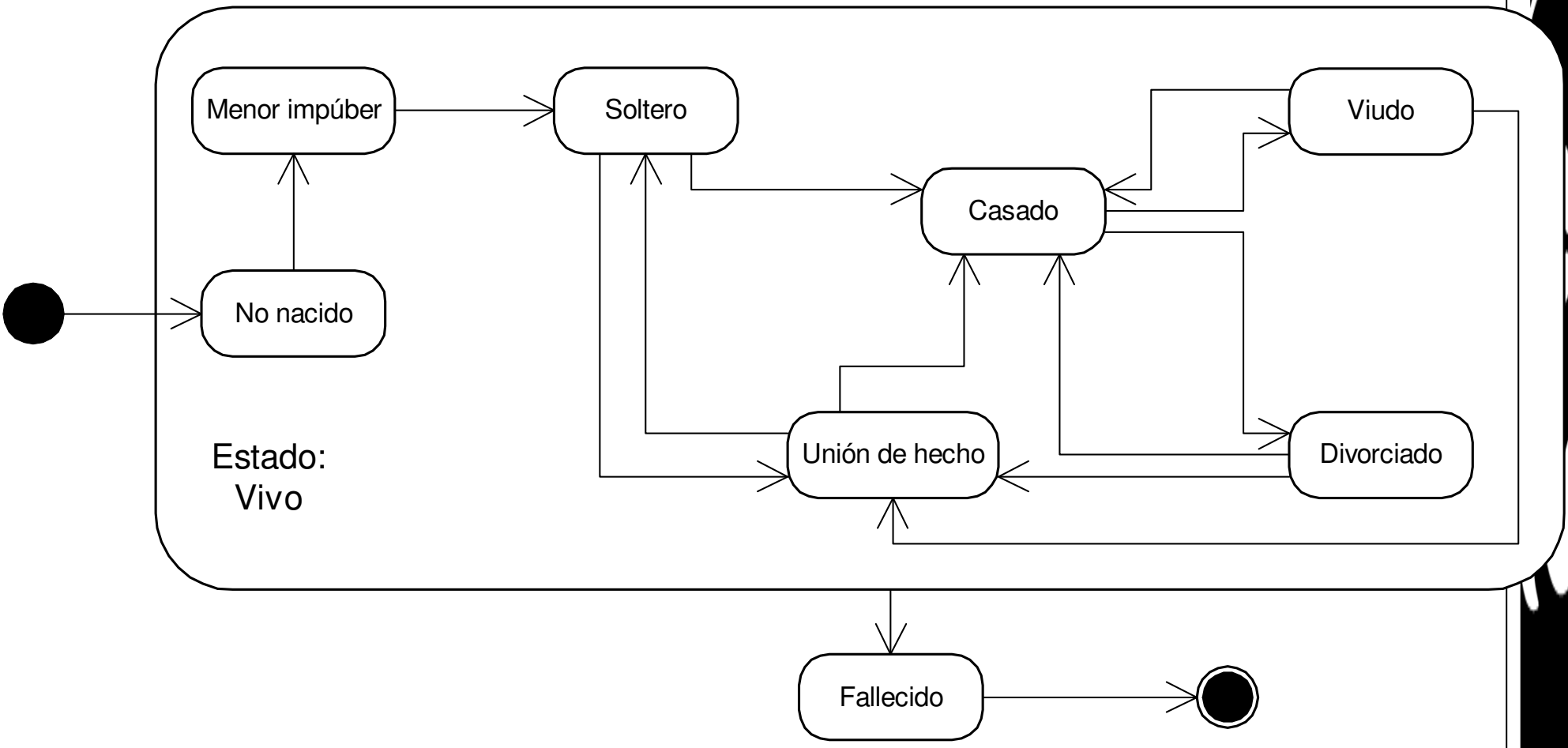
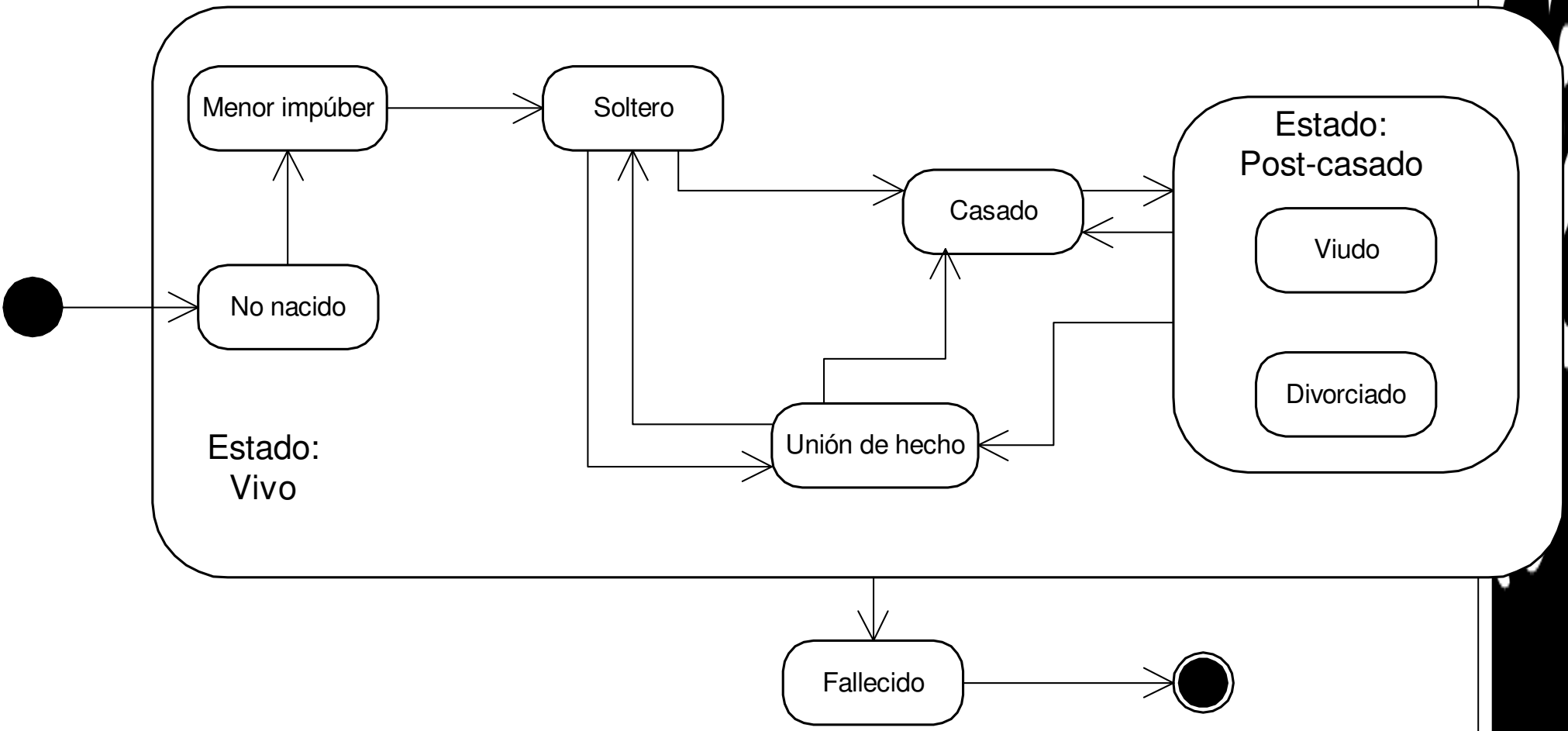


Diagrama de estados UML: estados civiles (2)



Claves

- Excepciones sirven para
  - Separar el código de tratamiento de problemas
  - Encarar la comprobación de estados con un enfoque optimista
- Colecciones e iteradores los vimos a modo de repaso de conceptos
  - Y de uso de principios de diseño que formalizaremos más adelante

Lectura obligatoria

"A Comparative Analysis of Generic Programming Paradigms in C++, Java and C#", Arijit Khan and Shatrugna Sadhu, http://www.cs.ucsb.edu/~arijitkhan/cs263.pdf

"Generics in C#, Java, and C++ - a conversation with Anders Hejlsberg, by Bill Venners with Bruce Eckel", http://www.artima.com/intv/genericsP.html

"Generics Considered Harmful", de Ken Arnold, http://weblogs.java.net/blog/arnold/archive/2005/06/generics_consid_1.html

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, capítulos 9 y 10 "Excepciones" y "Colecciones basadas en polimorfismo"

Qué sigue

Primer parcial

Código de calidad

Principios de diseño

MVC