# TEMA 3

## 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Excepciones

- Los objetivos que se deben buscar al diseñar un mecanismo de control de excepciones deben ser:
  - Evitar que la ocurrencia de excepciones se muestre al usuario final en pantalla de forma incontrolada.
  - No perder información de las excepciones lanzadas a más bajo nivel en la aplicación.
  - Mantener un Log de todas las excepciones que ocurren en el sistema y que sea lo más legible posible.
  - Dar soporte al sistema de validaciones de la aplicación.

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Excepciones

La estrategia planteada deberá ser en líneas generales la siguiente:

1. Diseñar una jerarquía de excepciones especificando en qué casos se debe utilizar cada una de ellas.
2. Establecer un conjunto de reglas para determinar que excepciones deben ser capturadas. Unas recomendaciones podrían ser:
   - Las excepciones controladas deben ser capturadas en el método en el que se producen, transformadas en las excepciones de aplicación adecuadas y enviadas hacia las capas superiores.
   - Las excepciones de tipo no controladas que se lancen en una capa deberán ser capturadas en la capa que la llama.
   - Las excepciones de tipo no controlado que se produzcan en la capas superiores de la aplicación e considerarán en general errores del sistema. Para evitar que su ocurrencia se muestre al .
   - En la interacción con el usuario se tendrán que capturar tanto las excepciones de tipo Aplicación que provienen de las capas inferiores como las que se produzcan en esta capa.
3. El sistema deberá gestionar los mensajes orientados a los usuarios y a los administradores. Los primeros serán los que se muestren en las páginas de la aplicación y los segundos los que se registren en el Log del sistema.

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Excepciones

- **ERROR**: es el resultado de un fallo o deficiencia durante el proceso de desarrollo y/o explotación de programas. Dicho fallo puede presentarse en cualquiera de las etapas del ciclo de vida del software.
- Tipos de Errores:
  - Errores de Compilación: producidos por el incumplimiento de las reglas léxicas, sintácticas y/o semánticas del lenguaje;
  - Errores de Ejecución:
    - Errores Lógicos: producidos por la lógica de un programa que no contempla todos los posibles valores de datos;
    - Errores Excepcionales: producidos por recursos (ficheros, comunicaciones, bibliotecas, …) fuera del ámbito del software que los maneja.

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Excepciones

**CONTEXTO** : ciertos errores lógicos (ej.: un valor negativo para calcular una factorial, una referencia sin la dirección del objeto -null-, …) pueden ser un error lógico o excepcional dependiendo del software en el que se está desarrollando:

- En el desarrollo de una aplicación se debe responsabilizar de la detección y subsanación de los errores lógicos dentro de su ámbito en la fase de pruebas.
- En el desarrollo de una biblioteca NO se puede responsabilizar del uso indebido de los servicios prestados a las aplicaciones y NUNCA debe responsabilizarse de la subsanación de dichos errores. En estos casos, estos errores lógicos se considerarán excepcionales porque la causa del error está fuera de los límites del software de la biblioteca.

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Aserciones

- **ASERCIÓN**: es una sentencia del lenguaje que permite comprobar las suposiciones del programa.
- Sintaxis:

```
assert <expresión1> [ : <expresión2> ] ;
```

- Cada aserción contiene una expresión lógica (<expresión>) que se supone cierta cuando se ejecute la sentencia. En caso contrario, el sistema finaliza la ejecución del programa y avisa del error detectado.
- Opcionalmente, se puede acompañar de una cadena de caracteres (<expresión2>) para detallar el error detectado.
- Escribir aserciones es una de las formas más rápidas y efectivas para detectar y corregir errores lógicos.
- Las aserciones permiten realizar comprobaciones de errores lógicos en la fase de pruebas, y eliminar estas comprobaciones automáticamente en el código de producción.

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Aserciones

- **ASERCIÓN**: es una sentencia del lenguaje que permite comprobar las suposiciones del programa.
- Sintaxis:

```
assert <expresión1> [ : <expresión2> ] ;
```

- Cada aserción contiene una expresión lógica (<expresión>) que se supone cierta cuando se ejecute la sentencia. En caso contrario, el sistema finaliza la ejecución del programa y avisa del error detectado.
- Opcionalmente, se puede acompañar de una cadena de caracteres (<expresión2>) para detallar el error detectado.
- Escribir aserciones es una de las formas más rápidas y efectivas para detectar y corregir errores lógicos.
- Las aserciones permiten realizar comprobaciones de errores lógicos en la fase de pruebas, y eliminar estas comprobaciones automáticamente en el código de producción.

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Aserciones

```
class Intervalo {
        …
        public Intervalo[] troceado(int veces) {
                Intervalo[] intervalos = new Intervalo[veces];
                double longitud = this.longitud() / veces;
                double minimo = this.minimo;
                double maximo = minimo + longitud;
                for (int i = 0; i < veces; i++) {
                        intervalos[i] = new Intervalo(minimo, maximo);
                        minimo = maximo;
                        maximo += longitud;
                }
                return intervalos;
        }
        …
}
```

```
class Aplicacion {
        GestorIO gestorIO = GestorIO.getGestorIO();
        Intervalo intervalo = new Intervalo();  intervalo.recoger();
        int veces = gestorIO.leerInt("Veces: ");
        Intervalo[] intervalos = intervalo.troceado(veces);
        …
}
```

¿Y si veces toma un valor negativo?

- se produce un error de ejecución,
- se muestra por pantalla un informe del error producido (reserva de un vector cuyo tamaño es negativo), y
- finaliza la ejecución del programa:

```
Introduce el minimo: 1
Introduce el maximo: 5
Veces: -1
Exception in thread "main" java.lang.NegativeArraySizeException
        at Intervalo.valores(Intervalo.java:141)  at
        Aplicacion.main(Aplicacion.java:7)
```

¿Cómo resolverlo?

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Aserciones

Si se trata de una aplicación, la solución **NO** es que el método compruebe la corrección de su argumento y que, en el caso de que no sea correcto, devuelva una valor que indique que hay un error.

**¡No es su responsabilidad!**

```
class Intervalo {
    …
    public Intervalo[] troceado(int veces) {
        if (veces <= 0) {
            return null;
        } else {
            Intervalo[] intervalos = new Intervalo[veces];
            double longitud = this.longitud() / veces;
            double minimo = this.minimo;
            double maximo = minimo + longitud;
            for (int i = 0; i < veces; i++) {
                intervalos[i] = new Intervalo(minimo, maximo);
                minimo = maximo;
                maximo += longitud;
            }
        return intervalos;
    }
}
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Aserciones

¡Es la aplicación la que, al leer el argumento, es responsable de verificar su corrección y que, en el caso de que no sea correcto, no se invoque al método y se comunique al usuario!

```
class Aplicacion {
    …
    int veces;
    do {
        veces = gestorIO.leerInt("Veces: ");
        if (veces <= 0) then {
            gestorIO.escribirLinea("El numero de veces
            debe ser positivo");
        }
    } while (veces <= 0);
    Intervalo[] intervalos = intervalo.troceado(veces);
    …
}
```

```
class Intervalo {
    …
    public Intervalo[] troceado(int veces) {
        Intervalo[] intervalos = new Intervalo[veces];
        double longitud = this.longitud() / veces;
        double minimo = this.minimo;
        double maximo = minimo + longitud;
        for (int i = 0; i < veces; i++) {
            intervalos[i] = new Intervalo(minimo, maximo);
            minimo = maximo;
            maximo += longitud;
        }
        return intervalos;
    }
    …
}
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Aserciones

Si se desea comprobar en el método la corrección de su argumento, lo que debe incluirse es una aserción

```
class Intervalo {
    …
    public Intervalo[] troceado(int veces) {
        assert veces > 0 : "El numero de veces no es positivo";
        Intervalo[] intervalos = new Intervalo[veces];
        double longitud = this.longitud() / veces;
        double minimo = this.minimo;
        double maximo = minimo + longitud;
        for (int i = 0; i < veces; i++) {
            intervalos[i] = new Intervalo(minimo, maximo);
            minimo = maximo;
            maximo += longitud;
        }
        return intervalos;
    }
    …
}
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Aserciones

Cuando se desarrolla código para una biblioteca, que va a ser utilizada desde aplicaciones, entonces **SI** que deberemos responsabilizarnos de comprobar los argumentos que se reciben en los métodos, porque están fuera del ámbito de la biblioteca.

Esto se consigue con la **gestión de excepciones.**

```
class Intervalo {
    …
    public Intervalo[] troceado(int veces) {
        // Gestión de excepciones
        Intervalo[] intervalos = new Intervalo[veces];
        double longitud = this.longitud() / veces;
        double minimo = this.minimo;
        double maximo = minimo + longitud;
        for (int i = 0; i < veces; i++) {
            intervalos[i] = new Intervalo(minimo, maximo);
            minimo = maximo;
            maximo += longitud;
        }
        return intervalos;
    }
    …
}
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Excepciones

- los errores en tiempo de ejecución se denominan en Java excepciones.
- las excepciones proporcionan una forma clara de indicar la existencia de posibles errores de ejecución, así como de comprobar la ocurrencia de dichos errores de ejecución sin oscurecer el código del programa.
- cuando se produce un error de ejecución en un programa, se **interrumpe** bruscamente la **ejecución** secuencial de sus instrucciones, es decir, no se ejecuta ninguna operación posterior a aquélla en la que se ha producido el error, y se **eleva** (o lanza) una excepción que representa ese error.
- por defecto, se muestra por pantalla un informe del error producido, y finaliza la ejecución del programa.

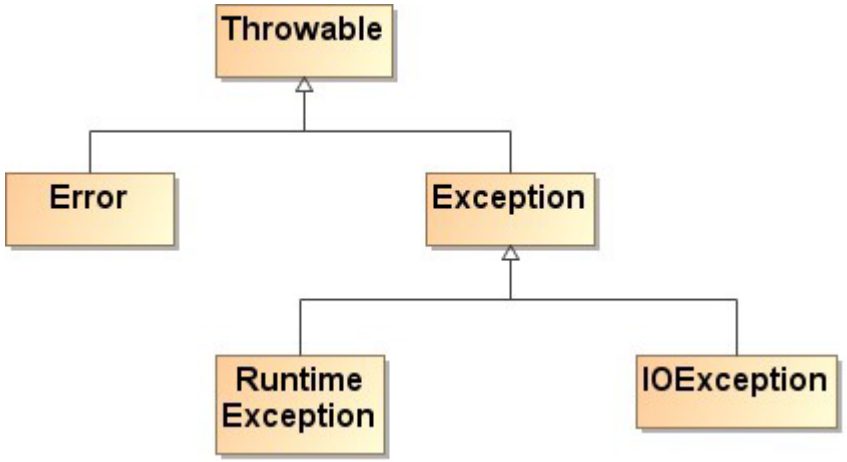
# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Gestión de Excepciones

- **elevar** la excepción: cuando una biblioteca encuentra un error porque no se cumplen las condiciones de su uso.
- **capturar** la excepción: cuando una aplicación o biblioteca usa un recurso (ficheros, comunicaciones, bibliotecas, …) que puede dar problemas, hay que capturarlos para solventar esos problemas.
- **delegar** la excepción:
  - cuando una aplicación o biblioteca usa un recurso (ficheros, comunicaciones, bibliotecas, …) que pueden dar problemas, se le comunica a quien corresponda para que los resuelva.
  - cuando una aplicación o biblioteca usa un recurso (ficheros, comunicaciones, bibliotecas, …) que pueden dar problemas, hay que capturarlos para solventar en parte esos problemas y elevar otra excepción a quien corresponda para que resuelva el resto.

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

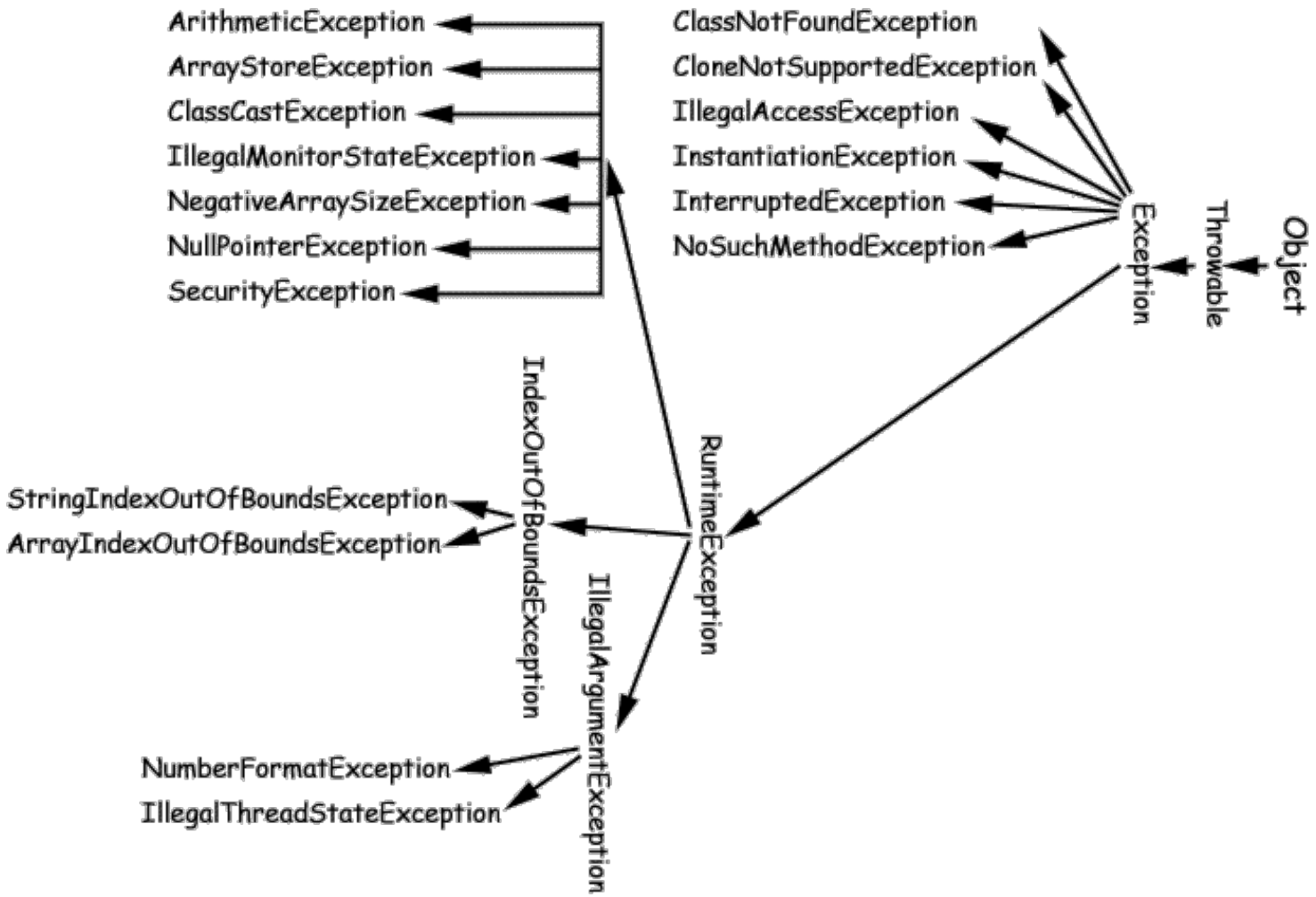
## Gestión de Excepciones



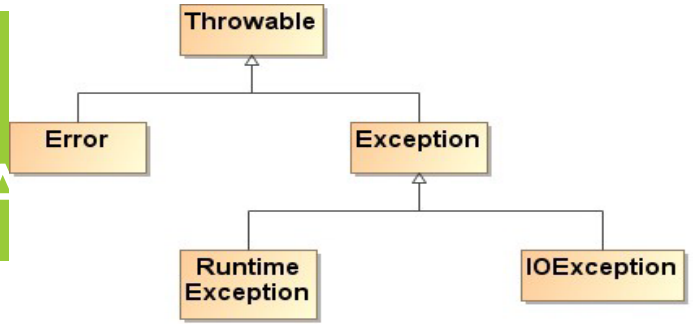

- las excepciones se representan en Java mediante objetos de clases que pertenecen a una jerarquía, cuyas clases más altas son:
- cuando se produce una excepción se crea un objeto de la clase adecuada, dependiendo del tipo de error producido, que mantendrá la información sobre el error y proporcionará métodos para obtener dicha información.
- la clase **Throwable** es la raíz de la jerarquía de clasificación y, por tanto, es la superclase de todas las excepciones de Java.

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Tipos de Excepciones



# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVA




## Gestión de Excepciones

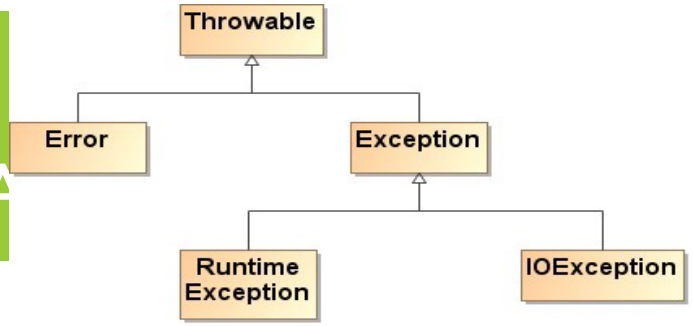
- La clase Error representa las excepciones graves que no deberían ser capturadas por un programa (por ejemplo, un error en la JVM).

```
java.lang.Object
     java.lang.Throwable
          java.lang.Error
               java.lang.VirtualMachineError
```

- La clase ***Exception*** representa las excepciones normales que un programa podría capturar.
- La clase ***RuntimeException*** representa aquellas excepciones normales que NO es necesario declarar en un programa (por ejemplo, una excepción aritmética de división por cero), estas excepciones se denominan "no comprobadas".

```
java.lang.Object
  java.lang.Throwable
     java.lang.Exception
        java.lang.RuntimeException
           java.lang.ArithmeticException
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVA




## Gestión de Excepciones

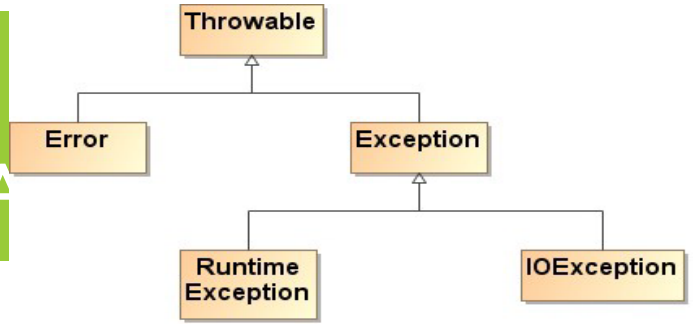
- El resto de clases que heredan de la clase **Exception** representan aquellas excepciones que SI es necesario capturar en un programa, estas excepciones se denominan "comprobadas".
- La clase **IOException** representa las excepciones comprobadas relacionadas con operaciones fallidas de entrada y salida, (por ejemplo, tratar de abrir en modo lectura un fichero inexistente),
- Algunos métodos de la clase **Throwable** son:
- public String getMessage()

  Devuelve un mensaje que informa de la excepción.
- public void printStackTrace()

  Imprime por la salida estándar los métodos que estaban en la pila de ejecución (llamadas anteriores a aquélla que produjo el error) cuando se produjo la excepción y la línea donde se produce la excepción. Es el comportamiento por defecto cuando no se maneja ninguna excepción.

```
java.lang.Objectjava.lang.Throwable
    java.lang.Exception
        java.io.IOException
            java.io.FileNotFoundException
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVA



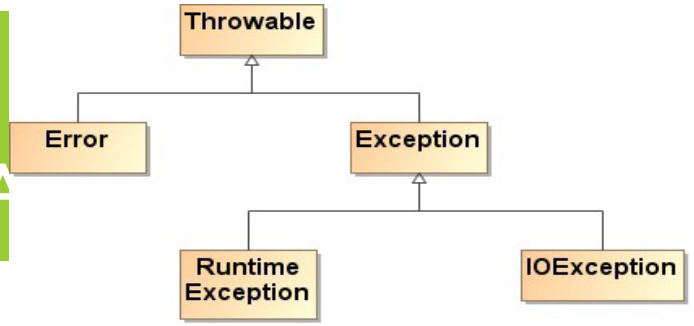

## Gestión de Excepciones

- Durante la ejecución de un método se puede elevar una excepción, para indicar que se ha producido un error de ejecución debido a alguna razón, lo que provoca la finalización brusca de su ejecución.
- Para elevar una excepción se debe usar la sentencia throw.

```
throw <expresión>;
```

- donde la expresión debe evaluar a una referencia a **Throwable**;
- en un mismo método se pueden elevar, de forma alternativa, varias excepciones;

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVA



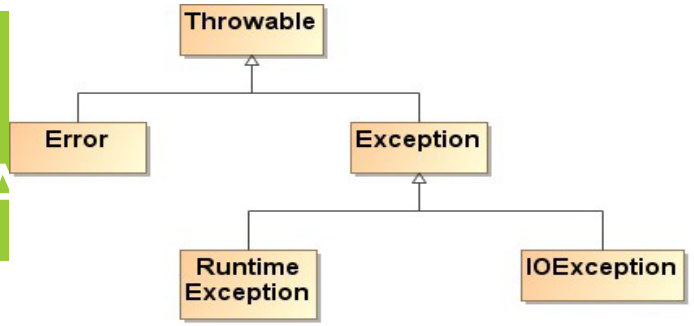

## Gestión de Excepciones

```
class Aplicacion {
   public static void main(String[] args) {
      GestorIO gestorIO = GestorIO.getGestorIO();
      gestorIO.escribirLinea("Antes");
      Fraccion fraccion = new Fraccion(3, 0);
      gestorIO.escribirLinea("Despues");
   }
}
```

```
// EN UNA BIBLIOTECA
public class Fraccion {
   private int numerador = 0;
   private int denominador = 1;
   public Fraccion(int numerador, int denominador) {
    if (denominador == 0)
        throw new ArithmeticException("El denominador no puede ser 0");
    this.numerador = numerador;
    this.denominador = denominador;
}
```

```
Resultado de la ejecución
Exception in thread "main" java.lang.ArithmeticException: El denominador no
puede ser 0
        at Fraccion.<init>(Fraccion.java:10)
        at Aplicacion.main(Aplicacion.java:6)
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVA




## Gestión de Excepciones

```
// EN UNA BIBLIOTECA

public class Fraccion {
    private int numerador = 0;
    private int denominador = 1;
    public Fraccion(String cadena) {
      if (!Pattern.matches(" *(\\d+) */ *(\\d+) *", cadena))
          throw new NumberFormatException("Formato incorrecto");
          Scanner scanner = new Scanner(cadena);
          int numerador = scanner.nextInt();
          scanner.next();
          int denominador = scanner.nextInt();
          if (denominador == 0)
                throw new ArithmeticException("El denominador no puede ser 0");
          this.numerador = numerador;
          this.denominador = denominador;
}
```

```
class Aplicacion {
    public static void main(String[] args) {
        GestorIO gestorIO = GestorIO.getGestorIO();
        gestorIO.escribirLinea("Antes de ejecutar");
        Fraccion fraccion = new Fraccion("3/2a");
        gestorIO.escribirLinea("Despues");
    }
}
```

```
Resultado de la ejecución
Antes de ejecutar
Exception in thread "main"
        java.lang.NumberFormatException: Formato
        incorrecto
        at Fraccion.<init>(Fraccion.java:17)
        at Aplicacion.main(Aplicacion.java:6)
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Gestión de Excepciones

- Las únicas excepciones comprobadas que puede elevar un método son aquéllas cuyas clases se declaran en su cabecera mediante la cláusula throws.

```
<modificador> <tipo1> <nombreMétodo>([<tipo2> <parametro>, …])
        throws <claseExcepción1>, … , <claseExcepciónN>
```

```
class Fraccion {
   …
   public void leer(String rutaFichero) throws IOException {
      …
      if (<error de lectura en el fichero>) {
         throw new IOException("Error al leer el fichero " + rutaFichero);
      } else {
         …
      }
   }
   …
}
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Gestión de Excepciones

- Un método puede elevar excepciones no comprobadas sin necesidad de declarar sus clases en la cabecera, aunque puede ser conveniente por razones de documentación.

```
public Fraccion(String cadena) throws NumberFormatException,
      ArithmeticException
   if (!Pattern.matches(" *(\\d+) */ *(\\d+) *", cadena))
      throw new NumberFormatException("Formato incorrecto");
   Scanner scanner = new Scanner(cadena);
   int numerador = scanner.nextInt();
   scanner.next();
   int denominador = scanner.nextInt();
   if (denominador == 0)
      throw new ArithmeticException("El denominador no puede ser 0");
   this.numerador = numerador;
   this.denominador = denominador;
}
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Captura de Excepciones

- Las excepciones no comprobadas elevadas por un método pueden ser capturadas en ese mismo método o en cualquier método que, directa o indirectamente, lo haya invocado.
- Si un método invoca otro método que eleva excepciones no comprobadas no es necesario que las capture, pero, en el caso de que se produzca la excepción, si ningún método captura la excepción, el programa termina bruscamente.
- Si un método invoca otro método que eleva excepciones comprobadas, entonces es obligatorio que el método que invoca al otro capture dichas excepciones.

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Captura de Excepciones

- Para capturar una excepción se debe usar la sentencia **try/catch/finally**
- Se puede omitir catch o finally, pero nunca ambas

```
try {
    <sentencia1>
    …
    <sentenciaN>
}
[
  catch (<declaraciónExcepción1>) {
    <sentencia11>
    …
    <sentencia1N1>
}
…
  catch (<declaraciónExcepciónM>) {
   …
}
]
[ finally {
   …
} ]
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Captura de Excepciones

- Bloque **try**:
  - Indica las sentencias en cuya ejecución se desea capturar una excepción, en el caso de que ésta se produzca.
  - Cuando se produce una excepción en alguna de sus sentencias, se interrumpe bruscamente su ejecución.
- Bloque(s) **catch**:
  - Especifican las sentencias a ejecutar cuando se produzca una excepción dentro del bloque try.
  - Se debe especificar la clase de excepción que queremos capturar mediante la declaración de una excepción de esa clase.
  - Opcionalmente, se pueden especificar distintos bloques catch, para indicar distintos tratamientos para distintas clases de excepciones.
  - Es un error especificar más de un bloque catch para una misma clase de excepción.

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Captura de Excepciones

- Bloque **finally**:
  - Se ejecuta siempre, después de la ejecución de try/catch.
  - Sirve para ejecutar un cierto código independientemente de si se produce o no la excepción, si se captura o no, o de cualquier otra circunstancia.
  - Se suele usar para garantizar la liberación de recursos, por ejemplo, cerrar ficheros archivos abiertos, puertos de comunicaciones, etc.

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Captura de Excepciones

```
class Aplicacion {
   public static void main(String[] args) {
      GestorIO gestorIO = GestorIO.getGestorIO();
      gestorIO.escribirLinea("Inicio");
      try {
         gestorIO.escribirLinea("Antes");
         Fraccion fraccion = new Fraccion(3, 0);
         gestorIO.escribirLinea("Despues");
      } catch (ArithmeticException ex) {
         gestorIO.escribirLinea("Error aritmetico");
         ex.printStackTrace();
      }
      gestorIO.escribirLinea("Fin");
   }
}
```

```
Inicio
Antes
Error aritmetico
java.lang.ArithmeticException: El denominador no puede ser 0
        at Fraccion.<init>(Fraccion.java:10)
        at Aplicacion.main(Aplicacion.java:10)
Fin
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Captura de Excepciones

```
class Aplicacion {
   public static void main(String[] args) {
      GestorIO gestorIO = GestorIO.getGestorIO();
      gestorIO.escribirLinea("Inicio");
      try {
         gestorIO.escribirLinea("Antes");
         Fraccion fraccion = new Fraccion("3/0");
         gestorIO.escribirLinea("Despues");
      } catch (ArithmeticException ex) {
         gestorIO.escribirLinea("Error aritmetico: " +
               ex.getMessage());
      } catch (NumberFormatException ex) {
         gestorIO.escribirLinea("Error de formato: " +
               ex.getMessage());
      }
      gestorIO.escribirLinea("Fin");
   }
}
```

```
Inicio
Antes
Error aritmetico: El denominador no puede ser 0
Fin
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Captura de Excepciones

```
class Aplicacion {
   public static void main(String[] args) {
      GestorIO gestorIO = GestorIO.getGestorIO();
      gestorIO.escribirLinea("Inicio");
      try {
         gestorIO.escribirLinea("Antes");
         Fraccion fraccion = new Fraccion("3/2a");
         gestorIO.escribirLinea("Despues");
      } catch (ArithmeticException ex) {
         gestorIO.escribirLinea("Error aritmetico: " +
               ex.getMessage());
      } catch (NumberFormatException ex) {
         gestorIO.escribirLinea("Error de formato: " +
               ex.getMessage());
      }
      gestorIO.escribirLinea("Fin");
   }
}
```

```
Inicio
Antes
Error de formato: Formato incorrecto
Fin
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Captura de Excepciones

- las excepciones no comprobadas se propagan automáticamente a través de las llamadas a métodos, siendo posible que la excepción se produzca en un método y se capture en otro que invoca, directa o indirectamente, a aquél en el que se produce.

```
try {
   gestorIO.escribirLinea("Antes");
   Fraccion fraccion = Fraccion.crearFraccion(3, 0);
   gestorIO.escribirLinea("Despues");
} catch (ArithmeticException ex) {
   gestorIO.escribirLinea("Error aritmetico");
   ex.printStackTrace();
}
gestorIO.escribirLinea("Fin");
…
public static Fraccion crearFraccion(int numerador, int denominador) {
   return new Fraccion(numerador, denominador);
}
```

```
Antes
Error aritmetico
java.lang.ArithmeticException: El denominador no puede ser 0
        at Fraccion.<init>(Fraccion.java:17)
        at Fraccion.crearFraccion(Fraccion.java:26)
        at Aplicacion.main(Aplicacion.java:14)
Fin
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Delegación de Excepciones

- Si un método declara que eleva excepciones comprobadas, entonces no necesita capturar esas clases de excepciones que declara, cuando invoca métodos que elevan esas mismas clases de excepciones.
- Si el método no captura tal excepción, la propaga al método que lo invocó, esto se conoce como delegación de excepciones;

```
class Calculadora {
        public Fraccion leerFraccion() throws IOException {
                Fraccion fraccion = new Fraccion();
                fraccion.leer("fichero.txt");
                return fraccion;
        }

}

class Fraccion {
        public void leer(String rutaFichero) throws IOException {
                …
        }
}
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Delegación de Excepciones

- Cuando un método delega una excepción, o la captura y eleva otra excepción, siempre se ejecuta el bloque finally.

```
class Fraccion {
    public void leer1(String rutaFichero) throws IOException {
        <abrir el fichero>
        try {
            <leer del fichero>
        } finally {
            <cerrar el fichero>
        }
    }
```

```
public void leer2(String rutaFichero) throws IOException {
    try {
        <abrir el fichero>
        try {
            <leer del fichero>
        } finally {
            <cerrar el fichero>
        }
    } catch (IOException ex) {
        <mostrar mensaje de error>  throw ex;
    }
}
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Excepciones y Polimorfismo

- Cuando un método captura una excepción, sólo se ejecuta la primera sentencia catch en la que la referencia de la excepción producida encaje en la clase de excepción declarada en esa sentencia catch.

```
gestorIO.escribirLinea("Inicio");
try {
        gestorIO.escribirLinea("Antes");
        Fraccion fraccion = new Fraccion("3/0");
        gestorIO.escribirLinea("Despues");
} catch (ArithmeticException ex) {
        gestorIO.escribirLinea("Error aritmético: " + ex.getMessage());
} catch (NumberFormatException ex) {
        gestorIO.escribirLinea("Error de formato: " + ex.getMessage());
} catch (Exception ex) {
        gestorIO.escribirLinea("Error generico: " + ex.getMessage());
        // NO SE EJECUTA
}
gestorIO.escribirLinea("Fin");
```

```
Inicio
Antes
Error aritmético: El denominador no puede ser 0
Fin
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Excepciones y Polimorfismo

- Los métodos pueden elevar excepciones de clases hijas de las declaradas en su cabecera mediante la cláusula **throws**.

```
public void m() throws Exception {
    …
    throw new IOException();
    …
}
```

- Los métodos redefinidos en una clase pueden eliminar excepciones de la cláusula **throws** de su cabecera de aquéllas que estaban declaradas en la cabecera del método en su clase padre.
- Los métodos redefinidos en una clase **NO** pueden añadir nuevas excepciones en la cláusula **throws** de su cabecera respecto a aquéllas que estaban declaradas en la cabecera del método en su clase padre.

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Excepciones y Polimorfismo

- Los métodos redefinidos en una clase pueden sustituir excepciones en la cláusula throws de su cabecera por otras excepciones de sus clases hijas;

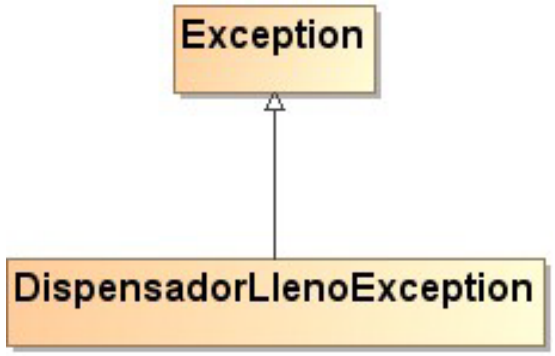
```
public class Base {
    public void m() throws Exception {…}
}

public class Derivada extends Base {
    public void m() throws IOException {…}
}
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Creación de Excepciones

- Es posible crear nuevas clases de excepciones siempre que hereden de la clase **Throwable** o de sus hijas, aunque, por convención, deben heredar de **Exception.**
- Estas clases pueden añadir nuevos atributos y métodos, como en cualquier otra relación de herencia.
- Ej. Al meter un elemento en un dispensador acotado que se encuentra lleno, se detecta un error de ejecución, que se puede representar mediante una nueva clase de excepción.



# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Creación de Excepciones

```
package pooa.dispensadores;
import pooa.util.GestorIO;
import pooa.util.Intervalo;
public class DispensadorLlenoException extends Exception {
   private DispensadorAcotado dispensador;
   private Intervalo intervalo;
   public DispensadorLlenoException(DispensadorAcotado dispensador, Intervalo intervalo) {
      super("El dispensador esta lleno");
      this.dispensador = dispensador;
      this.intervalo = intervalo;
   }
   public void reparar() {
      GestorIO.getGestorIO().escribirLinea("Reparando......");
      dispensador.duplicar();
      try {
         dispensador.meter(intervalo);
      } catch (DispensadorLlenoException ex) {}
   }
}
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Creación de Excepciones

```
public abstract class DispensadorAcotado implements Dispensador {
   …
   public void meter(Intervalo intervalo)
          throws DispensadorLlenoException {
      if (this.lleno()) {
         throw new DispensadorLlenoException(this, intervalo);
      } else {
         cuantos++;
         elementos[siguiente] = intervalo;
         siguiente++;
      }
   }
   …
   public abstract void duplicar();
}
```

```
public class PilaAcotada extends DispensadorAcotado {
   …
   public void duplicar() {
      Intervalo[] nuevos = new Intervalo[2 * elementos.length];
      for (int i = 0; i < cuantos; i++) {
         nuevos[i] = elementos[i];
      }
      elementos = nuevos;
      siguiente = cuantos;
   }
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Creación de Excepciones

```
public class ColaAcotada extends DispensadorAcotado {
   …
   public void duplicar() {
      Intervalo[] nuevos = new Intervalo[2 * elementos.length];
      int i = inicio;
      for (int j = 0; j < cuantos; j++) {
         nuevos[j] = elementos[i];
         i = (i + 1) % elementos.length;
      }
      elementos = nuevos;
      inicio = 0;
      siguiente = cuantos;
   }
```

```
   public static void main(String[] args) {
      …
      ColaAcotada cola = new ColaAcotada(tamaño);
      …
      try {
         cola.meter(new Intervalo(-i, i));
      } catch (DispensadorLlenoException ex) {
         ex.reparar();
      }
      …
   }
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

Objetivo

*Las clases parametrizadas están bien, pero, ¿y si quiero crear mi propia estructura?.*

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

***Recordar Tema 2.5.*** **Programación PARAMETRIZADA**

- Las clases parametrizadas se denominan clases genéricas; y los métodos parametrizados se denominan métodos genéricos.
- Los genéricos son útiles para definir clases cuyos atributos puedan ser de cualquier clase, y para definir métodos que puedan recibir argumentos y devolver resultados de cualquier clase.
- La mayoría de las estructuras básicas estándar ( colas, listas, pilas, … ) están ya implementadas como colecciones con este sistema y pueden usarse para mejorar el control y mantenimiento sin hacer uso de los vectores estándar ( array ).
- Estas ***colecciones*** ya tienen optimizados los procesos de inserción, ordenación, acceso y borrado por lo que es importante conocerlas y saber cual es la mas adecuada al problema y su necesidad para valorar que opción es la mas optima en cada caso.

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Programación **PARAMETRIZADA**

- Frente a esta solución, los genéricos tienen las siguientes ventajas:
  - proporcionan una comprobación estricta de tipos en tiempo de compilación;
  - su uso no necesita comprobación de tipos en tiempo de ejecución;
  - producen código más robusto y, en consecuencia, aumentan la facilidad del mantenimiento de los programas.

- Ejemplo
  - Se desea definir una clase PilaAcotada que pueda albergar objetos que sean de la misma clase, pero que esa clase pueda ser cualquiera;

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Ejemplo sin Parametrizada

Con esta solución es necesario convertir la dirección devuelta por el método *sacar()*, que es de la clase *Object*, a una dirección de la clase concreta mediante el operador de conversión de tipos (*cast*).

```
Ej.
public class PilaAcotada {
   private Object[] elementos;
   private int cuantos;
   public PilaAcotada(int tamaño) {
      elementos = new Object[tamaño];
      cuantos = 0;
   }
   public void meter(Object elemento) {
      elementos[cuantos++] = elemento;
   }
   public Object sacar() {
      return elementos[--cuantos];
   }
   public boolean vacia() {
      return cuantos == 0;
   }
   public boolean llena() {
      return cuantos == elementos.length;
   }
}
```

Al sacar un elemento existe la posibilidad de que se produzca un error de conversión de tipos en tiempo de ejecución al sacar los elementos, que no es detectado en tiempo de compilación, si los objetos que se meten en la pila son de distinta clase de la que se espera que sean al sacarlos.

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Ejemplo sin Parametrizada

```
public static void main(String[] args) {
    PilaAcotada pilaIntervalos = new PilaAcotada(10);
    Intervalo intervalo1 = new Intervalo(1, 2);
    pilaIntervalos.meter(intervalo1);
    Intervalo intervalo2 = new Intervalo(3, 4);
    pilaIntervalos.meter(intervalo2);
    intervalo1 = (Intervalo) pilaIntervalos.sacar();
    System.out.println(intervalo1);
    intervalo2 = (Intervalo) pilaIntervalos.sacar();
    System.out.println(intervalo2);
    System.out.println("  ");
    PilaAcotada pilaEnteros = new PilaAcotada(10);
    Integer i1 = new Integer(4);
    pilaEnteros.meter(i1);
    Integer i2 = new Integer(7);
    pilaEnteros.meter(i2);
    i1 = (Integer) pilaEnteros.sacar();
    System.out.println(i1);
    i2 = (Integer) pilaEnteros.sacar();
    System.out.println(i2);
    pilaEnteros.meter(intervalo1);
    i1 = (Integer) pilaEnteros.sacar();
    // ERROR DE EJECUCION
    System.out.println(i1);
}
```

```
[3, 4]
[1, 2]

7
4
Exception in thread "main" java.lang.ClassCastException:
     pooa.util1.Intervalo cannot be cast to java.lang.Integer
          at pooa.util1.PilaAcotada.main(PilaAcotada.java:55)
```

Estos problemas se pueden resolver mediante la declaración de una clase parametrizada, que permita declarar la clase de los elementos de la pila como un parámetro de tipo.

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Clases Genéricas Propias

- Sintaxis:

```
class <clase><<parámetro1>,…,<parámetroN>> {
    …
}
```

- donde los identificadores encerrados entre < y > son los parámetros de tipo.
- ***Los parámetros de tipo pueden usarse dentro de la clase parametrizada para declarar el tipo de sus atributos, argumentos de sus métodos, o objetos locales de sus métodos***.
- Estos parámetros de tipo (formales) se especifican posteriormente con un tipo concreto (actual) en la instanciación de un objeto de la clase parametrizada o al declarar una clase hija de la clase parametrizada, lo que produce la encarnación de la clase parametrizada, que se convierte en una clase concreta, proceso que se realiza en tiempo de compilación.

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Clases Genéricas Propias

Con esta solución **NO** es necesario convertir la dirección devuelta por el método *sacar()* a una dirección de la clase concreta mediante el operador de conversión de tipos (*cast*).

Ej.

```
public class PilaAcotada<Elemento> {
    private Elemento[] elementos;
    private int cuantos;

    public PilaAcotada(int tamaño) {
        elementos = (Elemento[]) new Object[tamaño];
        cuantos = 0;
    }

    public void meter(Elemento elemento) {
        elementos[cuantos++] = elemento;
    }

    public Elemento sacar() {
        return elementos[--cuantos];
    }

    public boolean vacia() {
        return cuantos == 0;
    }

    public boolean llena() {
        return cuantos == elementos.length;
    }
}
```

Al sacar un elemento **NO** existe la posibilidad de que se produzca un error de conversión de tipos en tiempo de ejecución al sacar los elementos, porque el error es detectado en tiempo decompilación, si se trata de meter elementos que no son de la clase adecuada.

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Clases Genéricas Propias

```
public static void main(String[] args) {
    // ENCARNACION DE LA CLASE PARAMETRIZADA
    PilaAcotada<Intervalo> pilaIntervalos =
              new PilaAcotada<Intervalo>(10);
    Intervalo intervalo1 = new Intervalo(1, 2);
    pilaIntervalos.meter(intervalo1);
    Intervalo intervalo2 = new Intervalo(3, 4);
    pilaIntervalos.meter(intervalo2);
    intervalo1 = pilaIntervalos.sacar();
    System.out.println(intervalo1);
    intervalo2 = pilaIntervalos.sacar();
    System.out.println(intervalo2);
    System.out.println("
");
    // ENCARNACION DE LA CLASE PARAMETRIZADA
    PilaAcotada<Integer> pilaEnteros = new PilaAcotada<Integer>(10);
    Integer i1 = new Integer(4);
    pilaEnteros.meter(i1);
    Integer i2 = new Integer(7);
    pilaEnteros.meter(i2);
    i1 = pilaEnteros.sacar();
    System.out.println(i1);
    i2 = pilaEnteros.sacar();
    System.out.println(i2);
    pilaEnteros.meter(intervalo1); // ERROR DE COMPILACION
    i1 = pilaEnteros.sacar();
    System.out.println(i1);
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Clases Genéricas Propias

En la definición de una clase pueden usarse clases parametrizadas

```
import java.util.ArrayList;

public class Bolsa < T > {
 private ArrayList < T > lista = new ArrayList < T >();

 public void add(T objeto){
    lista.add(objeto);
 }

 public ArrayList<T> getProducts(){
    return lista;
 }
}
```

```
public static void main(String[] args) {

    Bolsa<Caramelo> bolsaDeCaramelos = new Bolsa<Caramelo>();
    Bolsa<Chocolate> bolsaDeChocolates = new Bolsa<Chocolate>();

    bolsaDeCaramelos.add(new Soda("Caramelo_1", "Limon"));
    bolsaDeCaramelos.add(new Soda("Caramelo_2", "Fresa"));

    bolsaDeChocolates.add(new Chocolate("Chocolate_1", "Negro"));
    bolsaDeChocolates.add(new Chocolate("Chocolate_2", "Blanco"));

}
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Clases Genéricas Propias

- En la definición de una clase pueden usarse clases parametrizadas

Ej.

```
public class Lista<Elemento> {

    private Nodo<Elemento> inicio = null;
    private Nodo<Elemento> fin = null;
    private int longitud = 0;

    Nodo<Elemento> getInicio() {
        return inicio;
    }

    public int size() {
        return longitud;
    }

    public boolean isEmpty() {
        return inicio == null;
    }

    public Iterador<Elemento> iterador() {
        return new Iterador<Elemento>(this);
    }

    public Elemento get(int posicion) {
        Nodo<Elemento> nodo = inicio;
        for (int i = 0; i < posicion; i++) {
            nodo = nodo.next();
        }
        return nodo.getElemento();
    }

    public void add(Elemento elemento) {
        Nodo<Elemento> nuevo = new Nodo<Elemento>(elemento);
        if (fin == null) {
            inicio = nuevo;
        } else {
            fin.enlazar(nuevo);
        }
        fin = nuevo;
        longitud++;
    }
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Clases Genéricas Propias

```
public void remove(Elemento elemento) {
    Nodo<Elemento> anterior = null;
    Nodo<Elemento> actual = inicio;
    while (!actual.tienes(elemento)) {
        anterior = actual;
        actual = actual.next();
    }
    if (anterior == null) {
        inicio = actual.next();
    } else {
        anterior.enlazar(actual.next());
    }
    if (actual == fin) {
        fin = anterior;
    }
    longitud--;
}

public void clear() {
    inicio = null;
    fin = null;
    longitud = 0;
}
```

```
public Lista<Elemento> clone() {
    Lista<Elemento> lista = new Lista<Elemento>();
    if (!this.isEmpty()) {
        Iterador<Elemento> iterador = this.iterador();
        while (iterador.hasNext()) {
            lista.add(iterador.next());
        }
    }
    return lista;
}

public String toString() {
    String cadena = "{";
    Iterador<Elemento> iterador = this.iterador();
    if (iterador.hasNext()) {
        cadena += iterador.next();
    }
    while (iterador.hasNext()) {
        cadena += "; " + iterador.next();
    }
    return cadena + "}";
}
}
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Clases Genéricas Propias

```
class Nodo<Elemento> {

    private Elemento elemento;
    private Nodo<Elemento> siguiente;

    public Nodo(Elemento elemento) {
        this.elemento = elemento;
        this.siguiente = null;
    }

    public Elemento getElemento() {
        return elemento;
    }

    public void enlazar(Nodo<Elemento> nodo) {
        this.siguiente = nodo;
    }

    public boolean tienes(Elemento elemento) {
        return this.elemento == elemento;
    }

    public Nodo<Elemento> next() {
        return siguiente;
    }
}
```

```
public class Iterador<Elemento> {

    private Nodo<Elemento> actual;

    public Iterador(Lista<Elemento> lista){
        this.actual = lista.getInicio();
    }

    public boolean hasNext(){
        return actual != null;
    }

    public Elemento next() {
        Elemento elemento = actual.getElemento();
        actual = actual.next();
        return elemento;
    }
}
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Clases Genéricas Propias

```
public static void main(String[] args) {
    Integer uno = new Integer(1);
    Integer dos = new Integer(2);
    Integer tres = new Integer(3);
    Lista<Integer> listaEnteros = new Lista<Integer>();
    listaEnteros.add(uno);
    listaEnteros.add(dos);
    listaEnteros.add(tres);
    System.out.println(listaEnteros);

    Intervalo intervalo1 = new Intervalo(uno, dos);
    Intervalo intervalo2 = new Intervalo(uno, tres);
    Intervalo intervalo3 = new Intervalo(dos, tres);
    Lista<Intervalo> listaIntervalos1 = new Lista<Intervalo>();
    listaIntervalos1.add(intervalo1);
    listaIntervalos1.add(intervalo2);
    listaIntervalos1.add(intervalo3);
    System.out.println("-------------------------------------");
    System.out.println(listaIntervalos1);
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Clases Genéricas Propias

```
        intervalo1 = new Intervalo(tres, dos);
        intervalo2 = new Intervalo(tres, uno);
        intervalo3 = new Intervalo(dos, uno);
        Lista<Intervalo> listaIntervalos2 = new Lista<Intervalo>();
        listaIntervalos2.add(intervalo1);
        listaIntervalos2.add(intervalo2);
        listaIntervalos2.add(intervalo3);
        Lista<Lista<Intervalo>> listaListas =
              new Lista<Lista<Intervalo>>();
        listaListas.add(listaIntervalos1);
        listaListas.add(listaIntervalos2);
        System.out.println("-------------------------------------");
        System.out.println(listaListas);
    }
}
```

```
{1; 2; 3}
-------------------------------------
{[1, 2]; [1, 3]; [2, 3]}
-------------------------------------
{{[1, 2]; [1, 3]; [2, 3]}; {[3, 2]; [3, 1]; [2, 1]}}
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Clases Genéricas Propias (multi-tipo)

- Se pueden definir clases parametrizadas con varios parámetros de tipo.

Ej.

```
class Item<Clave, Valor> {
   private Clave clave;
   private Valor valor;

   Item(Clave clave, Valor valor) {
      this.clave = clave;
      this.valor = valor;
   }

   Clave getClave() {
      return clave;
   }

   Valor getValor() {
      return valor;
   }

   void setValor(Valor valor) {
      this.valor = valor;
   }
}
```

```
public class Map<Clave, Valor> {

   private static final int CAPACIDAD = 8191;
   private Lista<Item<Clave, Valor>>[] listas =      // AVISO
         new Lista[CAPACIDAD];

   public Valor put(Clave clave, Valor valor) {
      int i = this.hash(clave);
      if (listas[i] == null) {
         listas[i] = new Lista<Item<Clave, Valor>>();
      } else {
         Iterador<Item<Clave, Valor>> iterador = listas[i].iterador();
         while (iterador.hasNext()) {
            Item<Clave, Valor> item = iterador.next();
            if (clave.equals(item.getClave())) {
               Valor antiguo = item.getValor();
               item.setValor(valor);
               return antiguo;
            }
         }
      }
      listas[i].add(new Item<Clave, Valor>(clave, valor));
      return null;
   }
```

## Clases Genéricas Propias (multi-tipo)

```
public Valor get(Clave clave) {
    int i = this.hash(clave);
    Lista<Item<Clave, Valor>> lista = listas[i];
    Iterador<Item<Clave, Valor>> iterador = lista.iterador();
    while (iterador.hasNext()) {
        Item<Clave, Valor> item = iterador.next();
        if (clave.equals(item.getClave())) {
            return item.getValor();
        }
    }
    return null;
}

private int hash(Clave clave) {
    int hash = clave.hashCode();
    if (hash < 0) {
        hash = Math.abs(hash + 1);
    }
    return hash % CAPACIDAD;
}
```

```
Enero = 31
Febrero = 28
Marzo = 31
Abril = 30
Mayo = 31
```

```
public static void main(String[] args) {
    Map<String, Integer> map = new Map<String, Integer>();
    map.put("Enero", new Integer(31));
    map.put("Febrero", new Integer(28);
    map.put("Marzo", new Integer(31));
    map.put("Abril", new Integer(30));
    map.put("Mayo", new Integer(31));

    System.out.println("Enero = " + map.get("Enero"));
    System.out.println("Febrero = " + map.get("Febrero"));
    System.out.println("Marzo = " + map.get("Marzo"));
    System.out.println("Abril = " + map.get("Abril"));
    System.out.println("Mayo = " + map.get("Mayo"));
}
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Clases Genéricas Propias (Limitaciones)

- No se pueden crear vectores de objetos de clases genéricas.
- No se pueden encarnar clases genéricas con tipos primitivos

```
public class PilaAcotada<Elemento> {

    protected Elemento[] elementos;
    …

    public PilaAcotada(int tamaño) {
        elementos = new Elemento[tamaño];              // ERROR
        elementos = (Elemento[]) new Object[tamaño];   // AVISO
        …
    }
}

PilaAcotada<Integer>[] pilas = new PilaAcotada<Integer>[10];    // ERROR
PilaAcotada<Integer>[] pilas = new PilaAcotada[10];             // AVISO
```

```
PilaAcotada<int> pilaEnteros;        // ERROR
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Clases Genéricas Propias (Limitaciones)

- Limitaciones
  - **No se pueden crear clases parametrizadas de excepciones.**

```
public class MiExcepcion<Tipo> extends Exception {…}        // ERROR
```

  - **No pueden aparecer genéricos en las cláusulas catch**

```
try {…} catch (E e) {…}    // ERROR
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Clases genéricas (Comportamiento con la herencia )

- Una clase genérica puede heredar de otra clase genérica.

```
public class PilaAcotada<Elemento> extends DispensadorAcotado<Elemento>
```

- Una clase genérica puede heredar de una clase no genérica.

```
public class PilaAcotada<Elemento> implements Serializable
```

- Una clase no genérica puede heredar de una clase genérica.

```
public class PilaEnteros extends PilaAcotada<Integer>

public final class Integer extends Number implements Comparable<Integer>
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Parámetros limitados

- En la declaración de un parámetro de tipo de una clase parametrizada se puede especificar una restricción que debe cumplir ese tipo; estos parámetros de tipo se conocen como parámetros de tipo limitados (bounded).

```
class <clase><<parámetro> extends <tipoBase>,…> {
    …
}
```

- donde se especifica que el parámetro de tipo que se declara <parámetro> debe ser el tipo <tipoBase> o cualquier tipo derivado de él;
  - el tipo base puede ser una clase o un interfaz;
  - se pueden especificar más interfaces separándolos por &; <T extends B1 & B2 & B3>
  - el tipo base puede estar, a su vez, parametrizado.

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Parámetros limitados

- Por ejemplo, la clase Intervalo puede parametrizarse, pero con la restricción de que sus elementos puedan compararse entre sí;

Ej.

```
public class Intervalo<Elemento extends Comparable<Elemento>> {

    private Elemento minimo;
    private Elemento maximo;

    protected Elemento getMinimo() {
        return minimo;
    }

    protected Elemento getMaximo() {
        return maximo;
    }

    public Intervalo(Elemento minimo, Elemento maximo) {
        this.minimo = minimo;
        this.maximo = maximo;
    }

    public Intervalo(Intervalo<Elemento> intervalo) {
        this(intervalo.minimo, intervalo.maximo);
    }
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Parámetros limitados

- Por ejemplo, la clase Intervalo puede parametrizarse, pero con la restricción de que sus elementos puedan compararse entre sí;

```
public Intervalo<Elemento> copia() {
    return new Intervalo<Elemento>(this);
}

public boolean incluye(Elemento elemento) {
    return minimo.compareTo(elemento) <= 0 &&// LOS ELEMENTOS
           maximo.compareTo(elemento) >= 0;  // DEBEN SER COMPARABLES
}

public boolean incluye(Intervalo<Elemento> intervalo) {
    return this.incluye(intervalo.minimo) &&
           this.incluye(intervalo.maximo);
}

public boolean iguales(Intervalo<Elemento> intervalo) {
    return minimo.equals(intervalo.minimo) &&
           maximo.equals(intervalo.maximo);
}

public boolean distintos(Intervalo<Elemento> intervalo) {
    return !this.iguales(intervalo);
}
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Parámetros limitados

- Por ejemplo, la clase Intervalo puede parametrizarse, pero con la restricción de que sus elementos puedan compararse entre sí;

```
public String toString() {
    return "[" + minimo + ", " + maximo + "]";
}

public Intervalo<Elemento> interseccion(
        Intervalo<Elemento> intervalo) {
    if (this.incluye(intervalo)) {
        return intervalo.copia();
    } else if (intervalo.incluye(this)) {
        return this.copia();
    } else if (this.incluye(intervalo.minimo)) {
        return new Intervalo<Elemento>(intervalo.minimo, this.maximo);
    } else if (this.incluye(intervalo.maximo)) {
        return new Intervalo<Elemento>(this.minimo, intervalo.maximo);
    } else {
        return null;
    }
}
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Parámetros limitados

- Por ejemplo, la clase Intervalo puede parametrizarse, pero con la restricción de que sus elementos puedan compararse entre sí;

```
public static void main(String[] args) {
    Intervalo<Integer> intervaloEnteros1 = new Intervalo<Integer>(
            new Integer(1), new Integer(5));
    Intervalo<Integer> intervaloEnteros2 = new Intervalo<Integer>(
            new Integer(2), new Integer(3));
    System.out.println("El intervalo " + intervaloEnteros1 +
            (intervaloEnteros1.incluye(intervaloEnteros2) ? " SI" : " NO‟) +
            " incluye al intervalo " + intervaloEnteros2);

    Intervalo<Double> intervaloReales1 = new Intervalo<Double>(
            new Double(3.3), new Double(5.5));
    Intervalo<Double> intervaloReales2 = new Intervalo<Double>(
            new Double(4.4), new Double (6.6));
    System.out.println("El intervalo " + intervaloReales1 +
            (intervaloReales1.incluye(intervaloReales2) ? " SI" : " NO") +
            " incluye al intervalo " + intervaloReales2);
}
}
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Parámetros limitados

Ej.

```
public class Fecha implements Comparable<Fecha> {

   private Integer dia;
   private Integer mes;
   private Integer año;

   public Fecha(Integer dia, Integer mes, Integer año) {
      this.dia = dia;
      this.mes = mes;
      this.año = año;
   }

   public boolean iguales(Fecha fecha) {
      return dia.equals(fecha.dia) &&
            mes.equals(fecha.mes) &&
            año.equals(fecha.año);
   }

   public boolean distintas(Fecha fecha) {
      return !this.iguales(fecha);
   }
```

```
public String toString() {
   return dia + "/" + mes + "/" + año;
}

public int compareTo(Fecha fecha) {
   if (año == fecha.año) {
      if (mes == fecha.mes) {
         return dia.compareTo(fecha.dia);
      } else {
         return mes.compareTo(fecha.mes);
      }
   } else {
      return año.compareTo(fecha.año);
   }
}
```

## Parámetros limitados

```
public static void main(String[] args) {
    Fecha fecha11 = new Fecha(
          new Integer(1), new Integer(1), new Integer(1982));
    Fecha fecha12 = new Fecha(
          new Integer(31), new Integer(12), new Integer(1982));
    Intervalo<Fecha> intervaloFechas1 = new Intervalo<Fecha>(
          fecha11, fecha12);

    Fecha fecha21 = new Fecha(
          new Integer(1), new Integer(4), new Integer(1982));
    Fecha fecha22 = new Fecha(
          new Integer(30), new Integer(4), new Integer(1982));
    Intervalo<Fecha> intervaloFechas2 = new Intervalo<Fecha>(
          fecha21, fecha22);

    System.out.println("El intervalo " + intervaloFechas1 +
       (intervaloFechas1.incluye(intervaloFechas2) ? " SI" : " NO") +
       " incluye al intervalo " + intervaloFechas2);
  }
}
```

```
El intervalo [1/1/1982, 31/12/1982] SI incluye
al intervalo [1/4/1982, 30/4/1982]
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Parámetros Comodines

- Los parámetros de tipos comodines (wildcards) son parámetros sin nombre que pueden usarse para declarar tipos parametrizados de los atributos, argumentos, objetos locales y valores devueltos de métodos de cualquier clase, parametrizada o no; y que representan únicamente que se trata de un tipo cualquiera desconocido.
- Se representan mediante el símbolo **?**
- Al ser anónimos no pueden usarse para referirse a él en el interior de la clase o método donde se declaran.
- Pueden limitarse como cualquier otro parámetro de tipo.
- Su principal utilidad consiste en que relajan el sistema de tipos de Java de forma que resulta más fácil asignar instancias de tipos genéricos.

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Parámetros Comodines

Ej.

```
public class Lista<Elemento extends Number> {
   …
   public double suma(Lista<Number> lista) {
      double suma = 0.0;
      Iterador<Elemento> iterador = this.iterador();
      while (iterador.hasNext()) {
         Elemento elemento = iterador.next();
         suma += elemento.doubleValue();
      }
      Iterador<Number> iteradorLista = lista.iterador();
      while (iteradorLista.hasNext()) {
         Number numero = iteradorLista.next();
         suma += numero.doubleValue();
      }
      return suma;
   }
```

```
public static void main(String[] args) {
   Integer i1 = new Integer(1);
   Integer i2 = new Integer(2);
   Integer i3 = new Integer(3);
   Lista<Integer> listaEnteros = new Lista<Integer>();
   listaEnteros.add(i1);
   listaEnteros.add(i2);
   listaEnteros.add(i3);

   Double d1 = new Double(1.1);
   Double d2 = new Double(2.2);
   Double d3 = new Double(3.3);
   Lista<Double> listaReales = new Lista<Double>();
   listaReales.add(d1);
   listaReales.add(d2);
   listaReales.add(d3);
   // ERROR DE COMPILACION
   double suma = listaEnteros.suma(listaReales);
   System.out.println("La suma de los elementos de " +
         listaEnteros + " y " + listaReales + " es " + suma);
}
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Parámetros Comodines

- El error de compilación se produce porque la clase Lista<Double> no es una subclase de Lista<Number> aunque Double sea una subclase de Number (¿por qué?).
- La declaración Lista<Number> sólo funciona con objetos de la misma clase (no de la misma jerarquía)
- La razón es que si fuera así, entonces las siguientes sentencias:

```
Lista<Double> listaReales = new Lista<Double>();
Lista<Number> listaNumeros = listaReales;   // ERROR
listaNumeros.add(new Integer(1));
Double doble = listaReales.get(0);
```

Harían que pudiéramos copiar una referencia a un entero en una referencia a un doble, lo cual es incorrecto.

## Parámetros Comodines

- Para resolver este error de compilación se debe usar un parámetro de tipo comodín para declarar el tipo del argumento de la función suma(), como se muestra a continuación:

Ej.

```
public class Lista<Elemento extends Number> {
    …
    public double suma(Lista<? extends Number> lista) {
        double suma = 0.0;
        Iterador<Elemento> iterador = this.iterador();
        while (iterador.hasNext()) {
            Elemento elemento = iterador.next();
            suma += elemento.doubleValue();
        }
        Iterador<? extends Number> iteradorLista = lista.iterador();
        while (iteradorLista.hasNext()) {
            Number numero = iteradorLista.next();
            suma += numero.doubleValue();
        }
        return suma;
    }
    …
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Parámetros Comodines

- Para resolver este error de compilación se debe usar un parámetro de tipo comodín para declarar el tipo del argumento de la función suma(), como se muestra a continuación:

```
public static void main(String[] args) {
    Integer i1 = new Integer(1);
    Integer i2 = new Integer(2);
    Integer i3 = new Integer(3);
    Lista<Integer> listaEnteros = new Lista<Integer>();
    listaEnteros.add(i1);
    listaEnteros.add(i2);
    listaEnteros.add(i3);

    Double d1 = new Double(1.1);
    Double d2 = new Double(2.2);
    Double d3 = new Double(3.3);
    Lista<Double> listaReales = new Lista<Double>();
    listaReales.add(d1);
    listaReales.add(d2);
    listaReales.add(d3);
    double suma = listaEnteros.suma(listaReales);    // CORRECTO
    System.out.println("La suma de los elementos de " +
          listaEnteros + " y " + listaReales + " es " + suma);
}
```

```
La suma de los elementos de {1; 2; 3} y {1.1; 2.2; 3.3} es 12.600000000000001
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Parámetros Comodines

- El error de compilación no se produce porque el uso del parámetro comodín relaja el sistema de tipos, de manera que las siguientes sentencias son correctas:

```
Lista<Double> listaReales = new Lista<Double>();
Lista<? extends Number> listaNumeros = listaReales;      // CORRECTO
```

- Sin embargo, usar el parámetro comodín impone que con la referencia no se puedan usar métodos de la clase parametrizada que pudieran producir una incompatibilidad de tipos:

```
listaNumeros.add(new Double(3));     // ERROR
Double x = listaNumeros.get(0);      // ERROR
```

- Aunque sí se pueden usar métodos que no puedan producir incompatibilidad de tipos:

```
Number x = listaNumeros.get(0);                 // CORRECTO
String cadena = listaNumeros.toString();        // CORRECTO
listaNumeros.clear();                           // CORRECTO
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Parámetros Comodines

```
public class Lista<Elemento extends Number> {
   …
   public void addAll(Lista<? extends Elemento> lista) {
      Iterador<? extends Elemento> iteradorLista = lista.iterador();
      while (iteradorLista.hasNext()) {
         Elemento elemento = iteradorLista.next();
         this.add(elemento);
      }
   }
   …
   public static void main(String[] args) {
      …
      Lista<Number> listaNumeros = new Lista<Number>();
      listaNumeros.addAll(listaEnteros);
      listaNumeros.addAll(listaReales);
      System.out.println("La lista resultado de unir " +
            listaEnteros + " y " + listaReales + " es " +
            listaNumeros);
      …
   }
```

```
La lista resultado de unir {1; 2; 3} y {1.1; 2.2; 3.3}
es {1; 2; 3; 1.1; 2.2; 3.3}
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Métodos Genéricos

- Los métodos también se pueden parametrizar, tanto si están en una clase parametrizada como si no lo están.
- Para parametrizar un método se declaran los parámetros de tipo antes de la declaración del tipo del valor devuelto por el método.

```
<acceso> <<parámetro1>,…,<parámetroN>>
<tipoDevuelto> <nombreMétodo>(<argumentos>) {
    …
}
```

- Los parámetros de tipo pueden usarse dentro del método parametrizado para declarar el tipo del valor devuelto por el método, de sus argumentos y de sus objetos locales.
- Estos parámetros de tipo (formales) se especifican posteriormente con un tipo concreto (actual) al invocar el método parametrizado, lo que produce la encarnación del método.

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Métodos Genéricos

```
Ej.
public class Lista<Elemento extends Number> {
    …
    public <Tipo extends Number> Lista<Tipo> mayores(Lista<Tipo> lista) {
        Lista<Tipo> mayores = new Lista<Tipo>();
        Iterador<Elemento> iterador = this.iterador();
        Iterador<Tipo> iteradorLista = lista.iterador();
        while (iterador.hasNext() && iteradorLista.hasNext()) {
            Elemento elemento = iterador.next();
            Tipo valor = iteradorLista.next();
            if (valor.doubleValue() > elemento.doubleValue()) {
                mayores.add(valor);
            }
        }
        return mayores;
    }
    …
```

# 3.5. PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

## Métodos Genéricos

```
public static void main(String[] args) {
    Integer i1 = new Integer(1);
    Integer i2 = new Integer(2);
    Integer i3 = new Integer(3);
    Lista<Integer> listaEnteros = new Lista<Integer>();
    listaEnteros.add(i1);
    listaEnteros.add(i2);
    listaEnteros.add(i3);

    Double d1 = new Double(0.5);
    Double d2 = new Double(2.7);
    Double d3 = new Double(3.1);
    Lista<Double> listaReales = new Lista<Double>();
    listaReales.add(d1);
    listaReales.add(d2);
    listaReales.add(d3);

    Lista<Double> mayoresReales = listaEnteros.mayores(listaReales);
    System.out.println("Los elementos de " + listaReales +
            " mayores que " + listaEnteros + " son " + mayoresReales);

    Lista<Integer> mayoresEnteros = listaReales.mayores(listaEnteros);
    System.out.println("Los elementos de " + listaEnteros +
            " mayores que " + listaReales + " son " + mayoresEnteros);
}
}
```

```
Los elementos de {0.5; 2.7; 3.1} mayores que {1; 2; 3} son {2.7; 3.1}
Los elementos de {1; 2; 3} mayores que {0.5; 2.7; 3.1} son {1}
```