



Tema 3. Elementos de la POO: Herencia y Tratamiento de Excepciones

Programación (PRG)

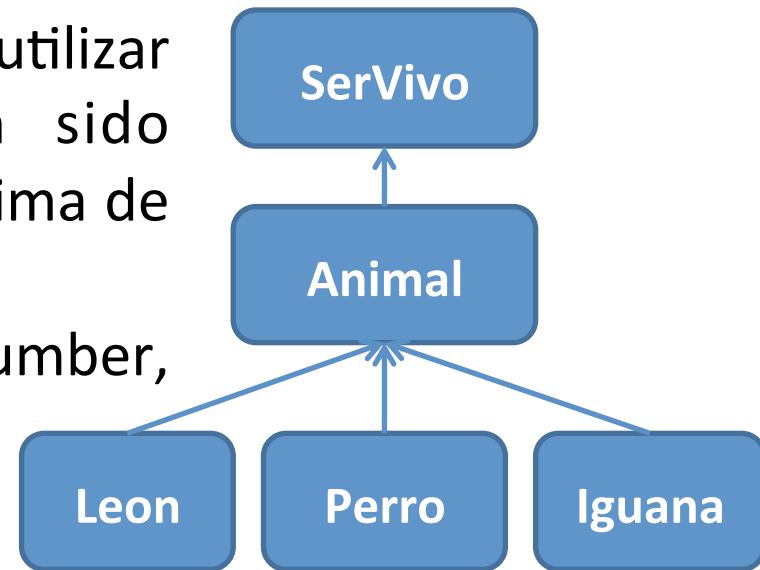
Departamento de Sistemas Informáticos y Computación

Contenidos

- Conceptos de la POO
 - Herencia
 - La jerarquía de clases en Java
 - La herencia en la documentación del API de Java
- Tratamiento de excepciones en Java
 - La jerarquía Throwable
 - Excepciones de usuario
 - Instrucción try-catch-finally
 - Instrucción throws
 - Instrucción throw

Introducción a la Herencia

- La herencia es el mecanismo que proporcionan los lenguajes de programación orientados a objetos para reutilizar el diseño de clases ya existentes para definir nuevas clases.
- Permite modelar de forma intuitiva una relación de tipo ES UN, definiendo una jerarquía de clases.
- Las clases pueden, de esta manera, utilizar atributos y métodos que hayan sido definidos en clases que estén por encima de ellas en la jerarquía.
- Ejemplo de la jerarquía de la clase Number, disponible en el API de Java.



Sobre la necesidad de la herencia

- Se pide construir la clase Estudiante con los atributos *nombre*, *edad*, y *créditos* matriculados. Ya se dispone de la clase Persona.

```
class Persona {  
    private String nombre;  
    private int edad;  
    public Persona(String nombre, int edad) {  
        this.nombre = nombre; this.edad = edad; }  
    public String getNombre() { return this.nombre; }  
    public int getEdad() { return this.edad; }  
    public String toString() {  
        return " Nombre: " + this.nombre + " Edad: " + this.edad; }  
}
```

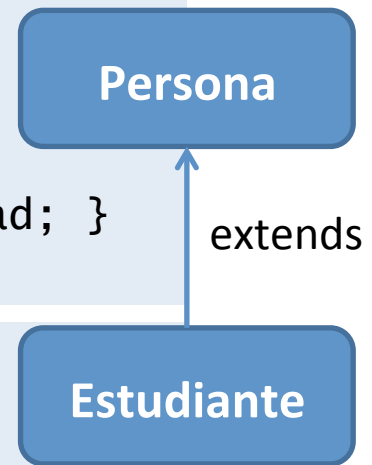
- Opciones posibles:
 - Inapropiada: Ignorar la clase Persona y construir la clase Estudiante con tres atributos (edad, nombre y créditos). Se repite la declaración de atributos y métodos ya realizado en la clase Persona.
 - Apropiada: Usar la herencia para definir la clase Estudiante en base a la clase Persona.

Construyendo una subclase

- Construcción de la clase derivada Estudiante a partir de la clase base Persona.
 - La clase Estudiante hereda (puede usar) todos los atributos y métodos que no son privados en Persona.

```
class Persona {  
    protected String nombre;  
    protected int edad;  
    public Persona(String nombre, int edad) {  
        this.nombre = nombre; this.edad = edad; }  
    public String getNombre() { return this.nombre; }  
    public int getEdad() { return this.edad; }  
    public String toString() {  
        return "Nombre: " + this.nombre + " Edad: " + this.edad; }  
}
```

```
class Estudiante extends Persona {  
    private int credits;  
    public Estudiante(String nombre, int edad) {  
        super(nombre, edad); this.credits = 60; }  
    public int getCredits() { return this.credits; }  
}
```



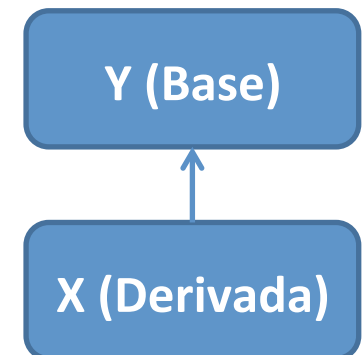
Utilizando una subclase

```
class TestEstudiantePersona {  
    public static void main(String[] args) {  
        Estudiante e = new Estudiante("Luisa Garcia",20);  
        Persona p = new Persona("Luisa Garcia",20);  
        System.out.println("Persona: " + p.toString());  
        System.out.println(e.getNombre() + " : " +  
                             e.getCreditos() + " créditos");  
    }  
}
```

- Se puede invocar a los métodos declarados en Persona desde un objeto Estudiante ya que Estudiante hereda de Persona.
 - La clase Estudiante puede acceder a los atributos y métodos no privados de la clase Persona.
- Muestra por pantalla:
Persona: Nombre: Luisa Garcia Edad: 20
Luisa Garcia: 60 créditos

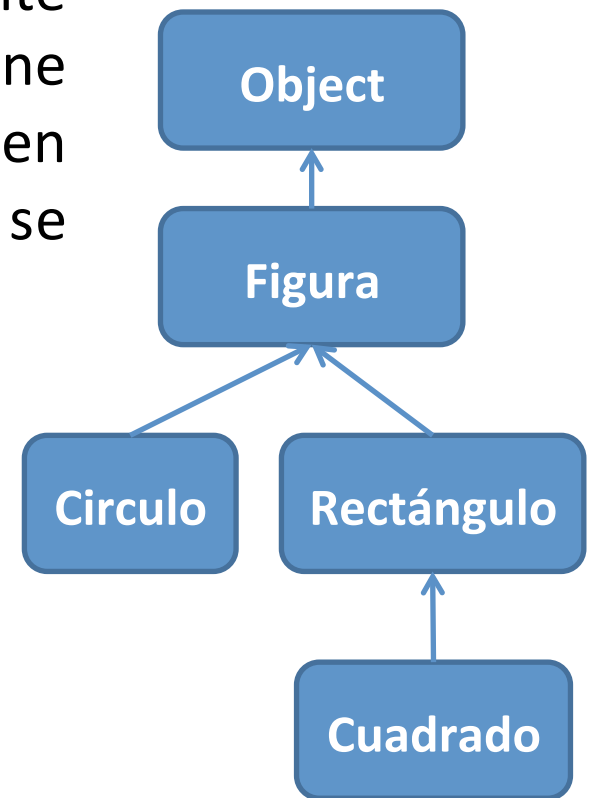
Sobre las jerarquías de clases

- Si X ES UN(A) Y,
 - Se dice que la clase derivada X es una variación de la clase base Y
 - Se dice que X e Y forman una jerarquía, donde la clase X es una subclase (o clase derivada) de Y e Y es una superclase (o clase padre) de X
 - La relación es transitiva: si X ES UN(A) Y e Y ES UN(A) Z, entonces X ES UN(A) Z
 - X puede referenciar todos los atributos y métodos que no sean privados en Y
 - X es una clase completamente nueva e independiente (los cambios en X no afectan a Y)
- Java no soporta herencia múltiple.
 - Una clase solo puede heredar como máximo de otra.



Jerarquías de Clases

- Cualquier clase Java hereda implícitamente de la clase predefinida Object. Ésta define los métodos que pueden ser invocados en cualquier objeto Java. Entre otros, se encuentran:
 - `public String toString()`
 - `public boolean equals(Object x)`
- Ejemplo de jerarquía de clases:
 - Una Figura ES UN Object
 - Un Círculo ES UNA Figura
 - Un Rectángulo ES UNA Figura
 - Un Cuadrado ES UN Rectángulo (cuya base y altura coinciden)



La herencia en la documentación de Java (I)

- Extracto de la documentación de la clase Number en el API Java.

java.lang

Class Number

[java.lang.Object](#)
└─ [java.lang.Number](#)

All Implemented Interfaces:
[Serializable](#)

Direct Known Subclasses:
[AtomicInteger](#), [AtomicLong](#), [BigDecimal](#), [BigInteger](#), [Byte](#), [Double](#), [Float](#), [Integer](#), [Long](#), [Short](#)

```
public abstract class Number
extends Object
implements Serializable
```

The abstract class Number is the superclass of classes ...

Subclasses of Number must provide methods to convert ...

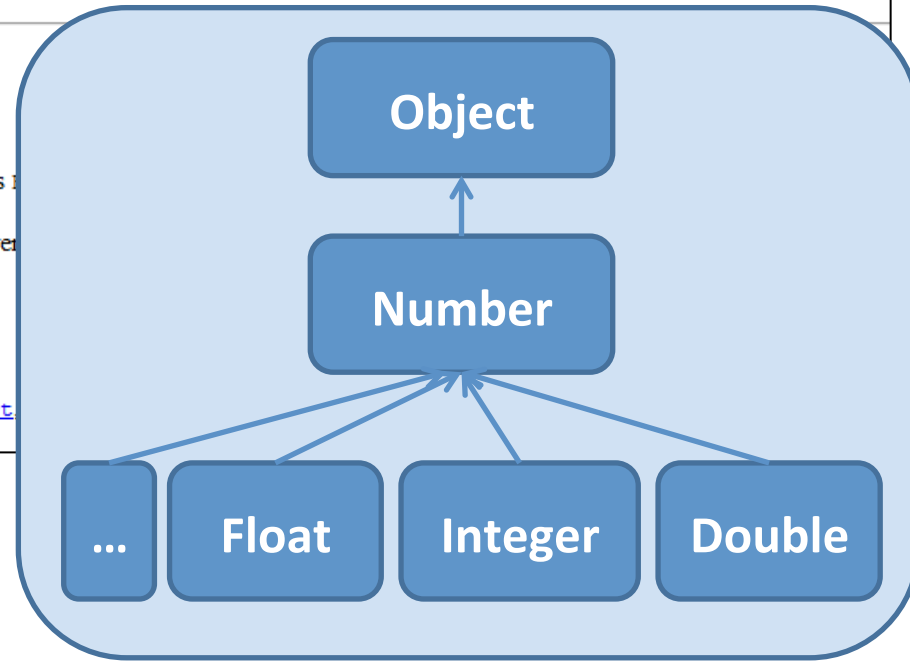
Since:
JDK1.0

See Also:
[Byte](#), [Double](#), [Float](#), [Integer](#), [Long](#), [Short](#)

La clase Number hereda de Object (como todas)

Subclases directas de la clase Number

Cabecera de la clase Number



La herencia en la documentación de Java (II)

- Extracto de la documentación de la clase Integer en el API Java.

java.lang

Class Integer

[java.lang.Object](#)
└ [java.lang.Number](#)
 └ java.lang.Integer

All Implemented Interfaces:
[Serializable](#), [Comparable<Integer>](#)

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

The Integer class wraps a value of the primitive type `int` in an object. An object of type `Integer` contains a single field whose type is `int`.

In addition, this class provides several methods for converting an `int` to a `String` and a `String` to an `int`, as well as other constants and methods useful when dealing with an `int`.

Implementation note: The implementations of the "bit twiddling" methods (such as [highestOneBit](#) and [numberOfTrailingZeros](#)) are based on material from Henry S. Warren, Jr.'s *Hacker's Delight*, (Addison Wesley, 2002).

Since:
JDK1.0

See Also:
[Serialized Form](#)

Jerarquía de la clase Integer (hereda de Number que a su vez hereda de Object)

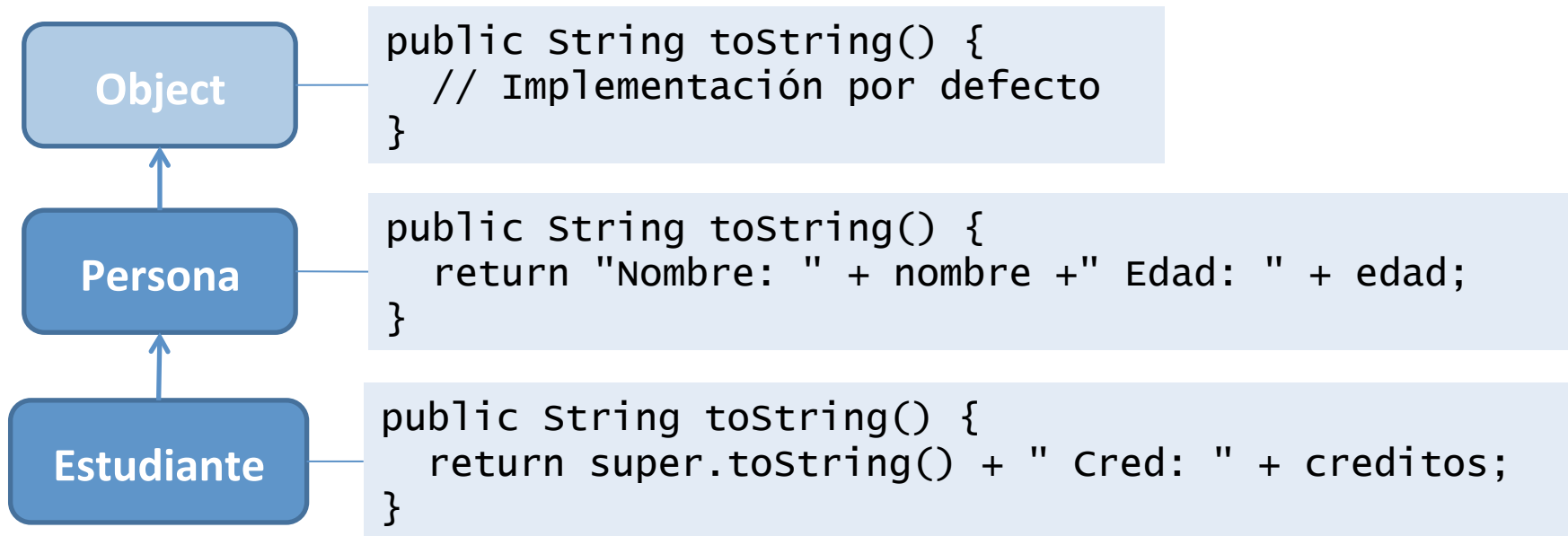
Cabecera de la clase Integer

Sobrescritura de métodos

- Cualquier método **no privado** de la clase Base que se defina de nuevo en la clase Derivada se **sobrescribe**:
- Sobrescritura completa:
 - Para ello, definimos en Derivada un método:
 - Con el mismo perfil que en Base (nombre y lista de parámetros).
 - Con el mismo tipo de resultado que en Base.
- Sobrescritura parcial:
 - Cuando tan solo se desea cambiar parcialmente el comportamiento del método de la clase Base. Se utiliza **super** para invocar el método de la clase Base.

Ejemplo de Sobrescritura (I)

- El caso más habitual suele ser la sobrescritura del método toString.



- La clase **Persona** sobrescribe completamente el método. La clase **Estudiante**, sobrescribe parcialmente el método.

Ejemplo de Sobrescritura (II)

```
class TestEstudiantePersona2 {  
    public static void main(String[] args) {  
        Estudiante e = new Estudiante("Luisa Garcia",20);  
        Persona p = new Persona("Luisa Garcia",20);  
        System.out.println("Persona: " + p);  
        System.out.println("Estudiante: " + e);  
    }  
}
```

- El programa muestra por pantalla:
Persona: Nombre: Luisa Garcia Edad: 20
Estudiante: Nombre: Luisa Garcia Edad: 20 Cred: 60
- No es necesario invocar explícitamente el método toString() del objeto, Java lo realiza automáticamente para poder concatenar con un String.

(Tratamiento de Excepciones en Java)

Introducción

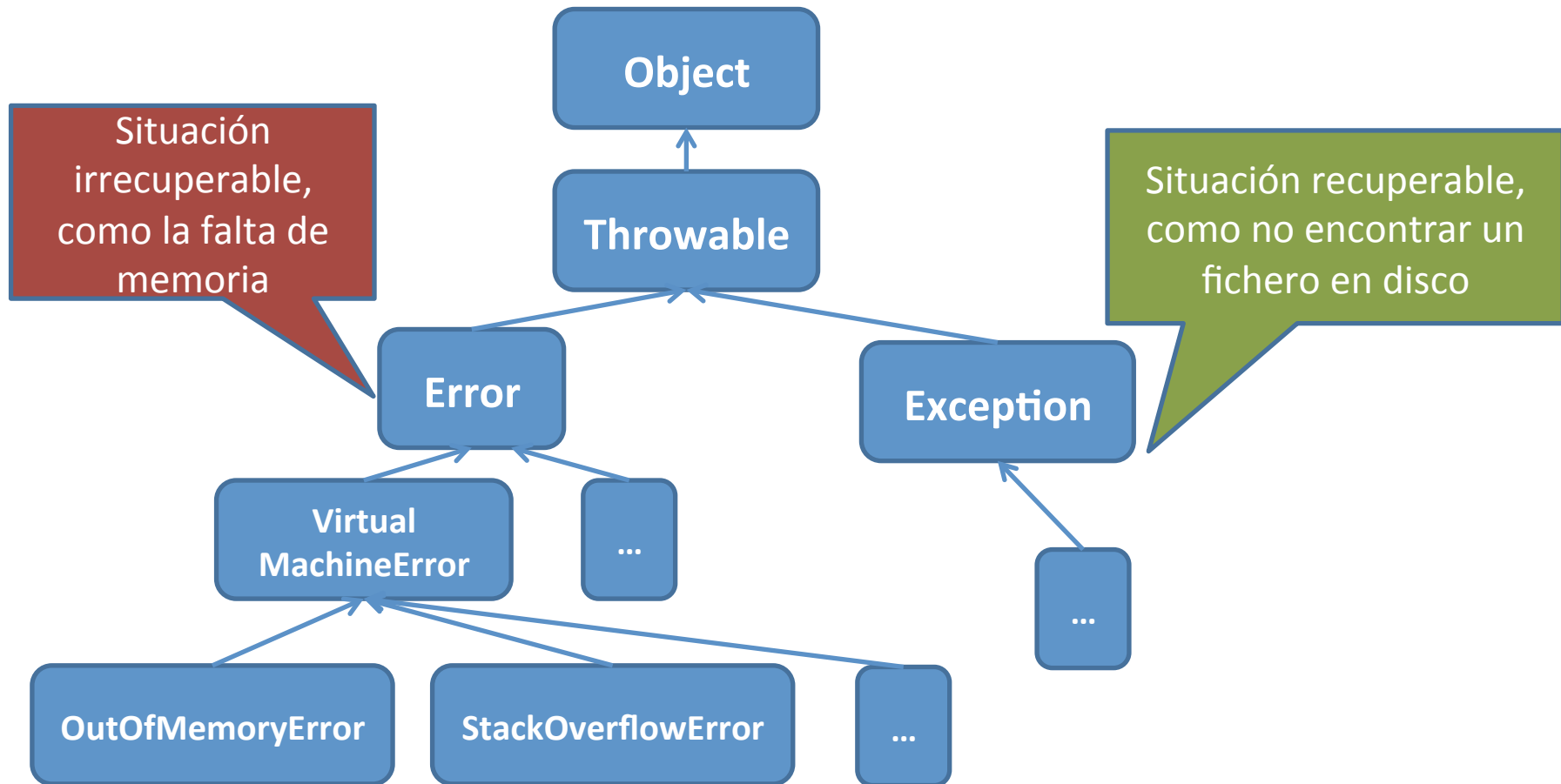
- ¿Qué ocurre cuando se produce un error inesperado, una excepción en el comportamiento de un programa que no ha sido prevista?
- Por ejemplo:
 - Ir a abrir un fichero y que no exista.
 - Pedir al usuario un número y que ponga una letra.
 - Invocar a un método de una referencia a null.
 - Acceder a una posición inexistente de un array.
 - Tratar de leer en un fichero más allá de su último dato.
- Alguno de estos errores son semánticos, otros no son culpa del programador. ¿Deberían estos errores provocar el fin de la ejecución del programa en todos los casos?
- Si no realizamos una gestión de las situaciones inesperadas, el programa abortará abruptamente.

Sobre las Situaciones Inesperadas

- Situaciones Inesperadas:
 - **Irrecuperables**
 - Agotamiento de memoria
 - Desbordar la pila de recursión
 - ...
 - **Recuperables**
 - Excepciones de Entrada/Salida
 - Excepciones de usuario
 - ...
- Las aplicaciones deben **prever y gestionar** las situaciones inesperadas que pueden ocurrir en ejecución.
 - Como mínimo **avisar** de la situación producida y, siempre que sea posible, tratar de **recuperarse** para que no afecte al normal desarrollo del programa.

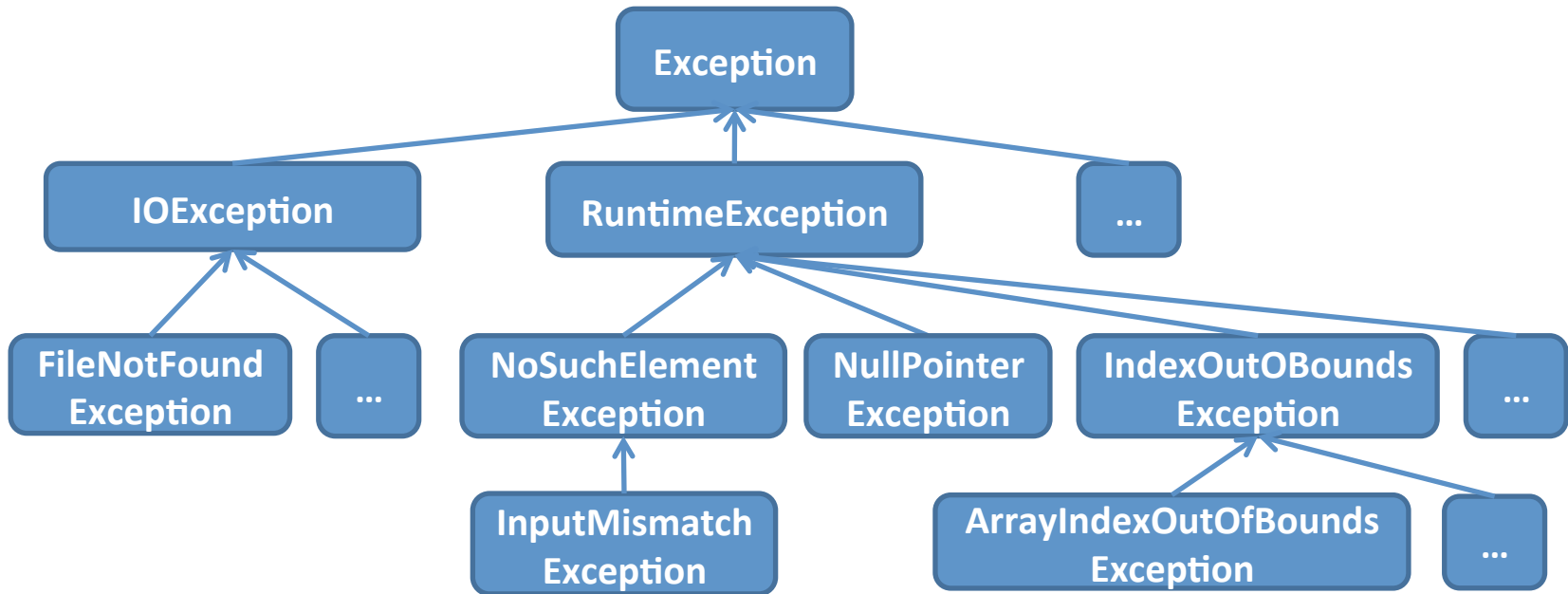
La Jerarquía Throwable

- Java incluye una jerarquía para representar los fallos de ejecución como objetos de subclases de Throwable.



La Jerarquía de Excepciones en Java

- Extracto de la jerarquía de excepciones en Java.



- Este tema se centra en la clase `Exception` y sus derivadas, en concreto:
 - `IOException`, para las excepciones de E/S de datos (como `FileNotFoundException`).
 - `RuntimeException` que representa las excepciones de fallos de programación (por ejemplo, `ArrayIndexOutOfBoundsException`).

Excepciones de Usuario

- También existe la posibilidad de definir excepciones de usuario, es decir, para representar fallos de la lógica de la aplicación, que el programador quiera gestionar.
 - Por ejemplo, se define un menú con opciones entre 0 y 5. Se podría definir una excepción de usuario (es decir, dentro de los paquetes que conforman la aplicación del usuario) denominada: “ExcepcionOpcionFueraDeRango” de la siguiente manera:

```
public class ExcepcionOpcionFueraDeRango extends Exception {  
    public ExcepcionOpcionFueraDeRango(String msg){ super(msg); }  
}
```

- Las excepciones de usuario son subclases de Exception y pueden incluir un mensaje descriptivo de la situación que condujo al fallo.

Definición de Excepciones de Usuario

- Las Excepciones de Usuario, que no pertenecen al API de Java, deben ser incorporadas como subclase de **Exception**.
- Ejemplo:
 - Aplicación de gestión de un grupo de Figuras donde se desea notificar el error al tratar de insertar en un grupo lleno.

```
public class GrupoLlenoException extends Exception {  
    public GrupoLlenoException(String mensaje) {  
        super(mensaje);  
    }  
    public GrupoLlenoException() { super(); }  
    public GrupoLlenoException(String s, Throwable t) {  
        super(s,t);  
    }  
}
```

Creación y Uso de una Nueva Excepción

- Las excepciones son objetos que deben crearse:

```
...  
GrupoLlenoException ex = new GrupoLlenoException();  
...
```

- Por lo general, los métodos deben indicar en su cabecera las excepciones que **pueden** lanzar:
 - Utilizando la clausula throws (separadas por comas).

```
...  
public void insertar(Figura f) throws GrupoLlenoException{  
... //En algún punto del código puede producirse la excepción  
}  
...
```

Ejemplo de Método que Lanza Excepción: parseInt de Integer

- El método parseInt de la clase Integer
 - Convierte una cadena que representa un número en un tipo primitivo int.
 - Si la cadena no representa un número (i.e. “45” vs “kjd”), lanza una excepción.

```
public class Integer {  
    ...  
    public static int parseInt(String s) throws NumberFormatException {  
        if (s == null) throw new NumberFormatException("null");  
        ... //Más adelante también puede lanzarse la excepción.  
    }  
}
```

Mecanismo de Gestión de Excepciones

- Invocar un método que lanza una excepción requiere una de estas gestiones:
 - **Capturar** la excepción (gestionándola), ó
 - **Propagar** la excepción (indicándolo en su definición), ó
 - **Capturar, gestionar y propagar** la excepción.
- Caso especial:
 - Las excepciones que son subclase de RuntimeException no tienen porqué declararse en la cabecera del método ni tienen porqué gestionarse obligatoriamente.
- La propagación de la excepción se hace **en orden inverso** a la secuencia de llamadas realizada.
 - Hasta que las gestiona algún método o la JVM aborta.

Tratamiento de excepciones en Java

- ¿Qué hacer con una excepción una vez se ha producido?

Las excepciones pueden ser lanzadas, propagadas y capturadas...

Un inciso: En realidad algunos tipos de excepciones **DEBEN** ser propagadas o capturadas y otras **PUEDEN** ser propagadas o capturadas. En concreto, **todas las derivadas directamente de Exception (excepto las RuntimeException) incluidas las de usuario (que no deriven de RuntimeException)** se denominan excepciones ***checked*** o comprobadas y siempre **se deben** propagar o capturar, mientras que las derivadas de **RuntimeException** se denominan ***unchecked*** o no comprobadas y **se pueden** capturar o propagar o simplemente dejar que aborten el programa.

Tratamiento de excepciones en Java

- La gestión por defecto de la excepción provoca la finalización abrupta del programa.
- Sin embargo, se podría hacer que el método capturase la excepción y continuase funcionando incluso con datos que provocarían un error.
- Java provee un mecanismo que permite gestionar las excepciones que se pueden provocar en un bloque de código. Se trata de la instrucción try-catch (ó try-catch-finally).

Captura de Excepciones

```
try {  
    // código donde pueden producirse las excepciones e1, e2, etc.  
}  
catch (ExcepcionTipo1 e1) {  
    // código de gestión de e1  
}  
catch (ExcepcionTipo2 e2) {  
    // código de gestión de e2  
}  
...  
}  
finally {  
    // código que se ejecuta siempre,  
    // tanto si el try se completa como si no  
}
```

- Ya que todas las excepciones son subclases de Exception, siempre se puede realizar un try-catch general de Exception y capturar cualquier tipo de excepción, pero no es recomendable, ya que se capturarían aquellas que se sabe que se pueden producir pero también aquellas con las que no se contaba, llevando a errores de ejecución de difícil detección.

Tratamiento de excepciones en Java

- **try:** contiene las instrucciones donde se puede producir uno o varios tipos de excepciones:
 - Si se produce una excepción, el flujo de ejecución salta directamente al catch de esa excepción (o al de una de sus superclases).
 - Si no se produce excepción, se ejecuta normalmente.
- **catch:** contiene las instrucciones a realizar si se produjo una excepción:
 - Debe haber al menos uno por cada try (si no hay finally).
 - Indica qué tipo de excepción se captura.
 - Si hubiera más de una excepción posible, habría más de un bloque catch.
- **finally:** contiene instrucciones que se ejecutan tanto si se completó el try como si no:
 - es opcional. Se usa por si se producen excepciones que no son comprobadas ni capturadas y que abortarían el programa pero que, aún así, haya que realizar determinadas acciones antes de terminar. Por ejemplo, guardar los datos y cerrar un fichero en disco.

Ejemplo de Excepciones: Captura

```
public class Transfer{  
    public void transferFile(File f) throws UnableToTransfer{ ... }  
}
```

```
public class Modulo{  
    Transfer t;  
    ...  
    public void copy(File f) {  
        ...  
        try{  
            t.transferFile(f);  
            ...  
        }catch(UnableToTransfer ex) {  
            System.err.println("Imposible transferir: " + ex.getMessage());  
        }  
        ...  
    }  
}
```

Ejemplo de Excepciones: Propagación

- Diseñar un módulo de transferencia (clase Modulo) que utilice la funcionalidad de Transfer.

```
public class Transfer{  
    public void transferFile (File f) throws UnableToTransfer{...}  
}
```

```
public class Modulo {  
    Transfer t;  
    ...  
    public void copy(File f) throws UnableToTransfer{  
        ...  
        t.transferFile(f);  
        ...  
    }  
}
```

Ejemplo de Excepciones: Captura y Propagación

```
public class Transfer{  
    public void transferFile (File f) throws UnableToTransfer{ ... }  
}
```

```
public class Modulo {  
    Transfer t;  
    ...  
    public void copy(File f) throws UnableToTransfer{  
    ...  
    try{  
        t.transferFile(f);  
    }catch(UnableToTransfer ex) {  
        System.out.println("Imposible transferir");  
        throw ex; //No es necesario throw new UnableToTransfer("..");  
    }  
}}
```


Tratamiento de excepciones en Java

- ¿Qué hacer con una excepción una vez se ha producido?

Una excepción se lanza cuando se produce el hecho que la provoca, por ejemplo, al tratar de abrir un fichero que no existe. Sin embargo también pueden ser creadas y lanzadas voluntariamente:

- Supongamos que el usuario introduce un valor al programa fuera del intervalo esperado (por ejemplo, un número mayor o igual que 60 o menor a cero en la variable minutos). Para lanzar la excepción se usa **throw**.

```
Scanner t = new Scanner(System.in);  
System.out.print("¿Minutos?: ");  
int minutos = t.nextInt();  
if (minutos<0 || minutos>=60)  
    throw new InputMismatchException("valor incorrecto");
```



Ejemplo de Excepciones (I): Propagación

- Diseñar un módulo de grabación (clase ModuloGrabacion) que utilice la funcionalidad de GrabacionDVD.

```
public class GrabacionDVD {  
    public void grabar() throws ExcepcionEnGrabacion { ... }  
}
```

```
public class ModuloGrabacion {  
    GrabacionDVD grabDVD;  
    ...  
    public void crearDVD() throws ExcepcionEnGrabacion {  
        ...  
        grabDVD.grabar();  
        ...  
    }  
}}
```


Ejemplo de Excepciones (II): Captura

```
public class GrabacionDVD {  
    public void grabar() throws ExcepcionEnGrabacion { ... }  
}
```

```
public class ModuloGrabacion {  
    GrabacionDVD grabDVD;  
    ...  
    public void crearDVD() {  
        try{  
            grabDVD.grabar();  
        }catch(ExcepcionEnGrabacion ex) {  
            System.out.println("Te he fastidiado un DVD");  
        }  
    }  
}
```

Ejemplo de Excepciones (III): Captura y Propagación

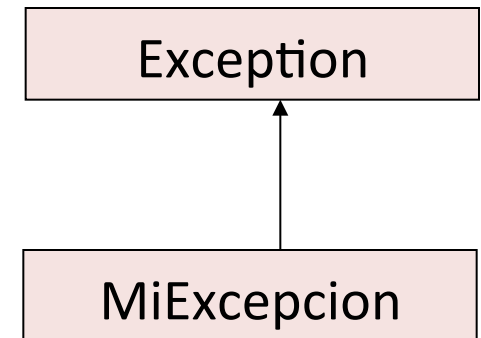
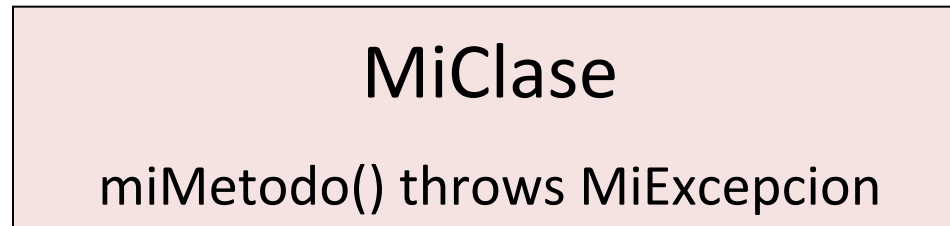
```
public class GrabacionDVD {  
    public void grabar() throws ExcepcionEnGrabacion { ... }  
}
```

```
public class ModuloGrabacion {  
    GrabacionDVD grabDVD;  
    ...  
    public void crearDVD() throws ExcepcionEnGrabacion {  
        try {  
            grabDVD.grabar();  
        } catch (ExcepcionEnGrabacion ex) {  
            System.out.println("Te he fastidiado un DVD");  
            throw ex; //No es necesario throw new ExcepcionEnGrabacion("..");  
        }  
    }  
}
```



Ejercicio Excepciones

- Dadas las siguientes clases:



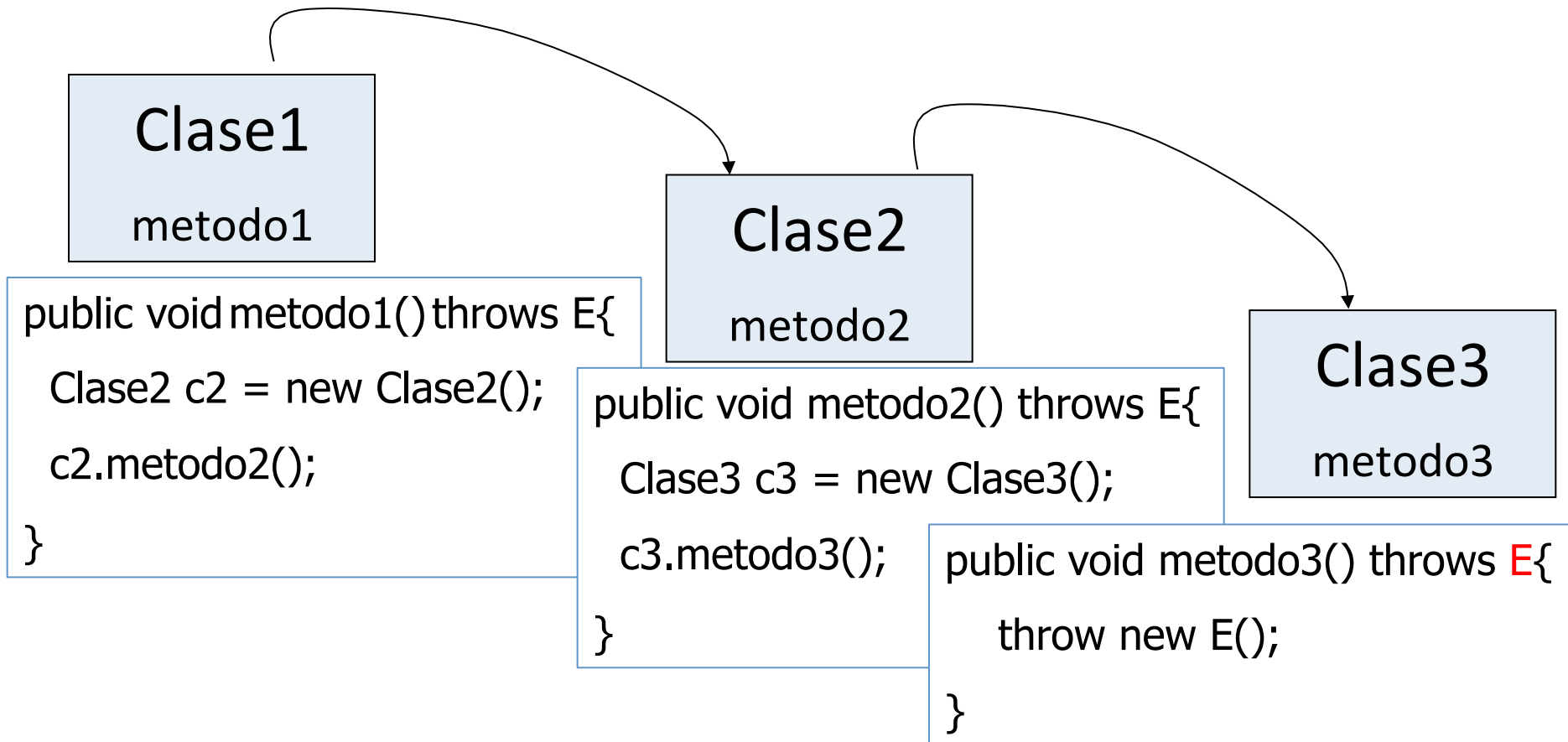
- ¿Compilaría el siguiente código?:

```
public static void main(String args[]){  
    MiClase mc = new MiClase();  
    mc.miMetodo();  
}
```

- Si no compila, realizar las modificaciones oportunas.

Propagación de Excepciones en Java

- ¿Cuál será el orden de propagación de la excepción E lanzada en método 3?



Tratamiento de excepciones en Java

- Con la cláusula **throws** se puede propagar la excepción por toda la pila de llamadas, situando el tratamiento allí donde se desee.
- Puede haber más de una excepción (separadas por comas) en una misma cláusula **throws**.
- No confundir la cláusula **throws** para propagar la excepción con **throw** para lanzar una excepción nueva. (Recordar que hay que tenerla creada, ya que es un objeto).
- Las excepciones checked siempre deben ser propagadas o capturadas.
- Al igual que con el resto de estructuras de control, las instrucciones try-catch-finally pueden ser anidadas o situadas dentro de bucles, condicionales, etc.

Ejemplo: Lectura Validada

- Lectura de un valor entero en [0,5]

```
import java.util.*;
public class LecturaValidada{
    public static void main(String args[]){
        Scanner teclado = new Scanner(System.in);
        int x = -1;
        do{
            try{
                System.out.println("Introduce un entero en [0,5]:");
                x = teclado.nextInt();
            }catch(InputMismatchException ex){
                teclado.next();
            }
        }while (x < 0 || x > 5);
        System.out.println("El valor de x es: " + x);
    }
}
```

El try-catch debe ir
DENTRO del bucle para
repetir la operación en
caso de entrada
incorrecta.

Conclusiones

- Java permite modelar las situaciones inesperadas en tiempo de ejecución como excepciones.
- Las excepciones son objetos cuya clase es subclase de Exception.
- Al invocar un método que puede lanzar excepciones, es necesario gestionarlas convenientemente:
 - Capturando, propagando o realizando ambas aproximaciones.
- El programador puede definir nuevas excepciones para usarlas en sus aplicaciones.