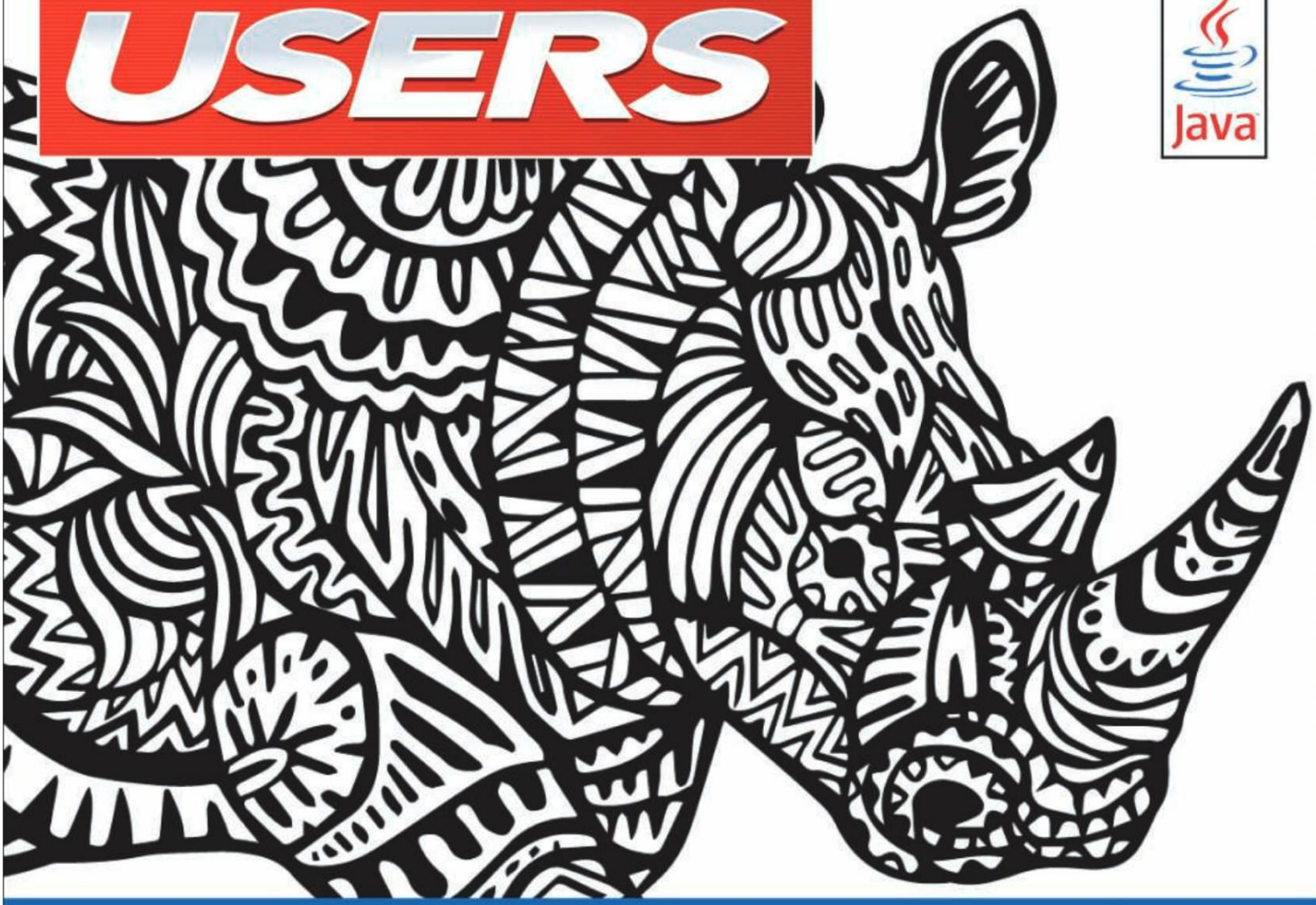
USERS



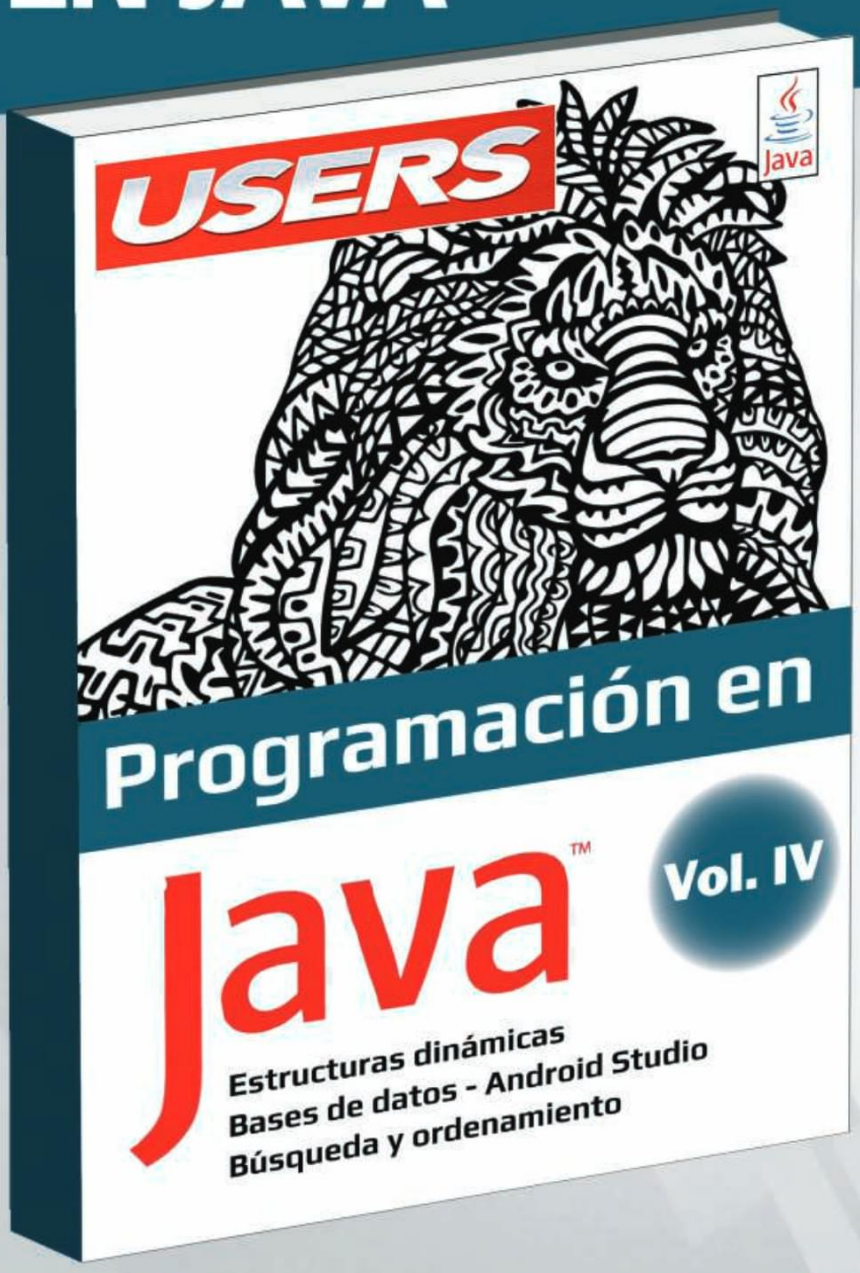
Programación en Java™

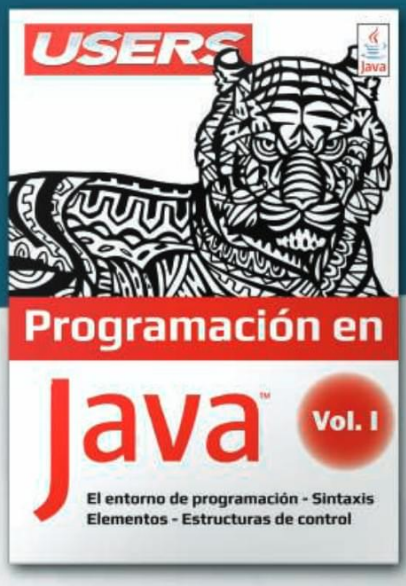
Vol. III

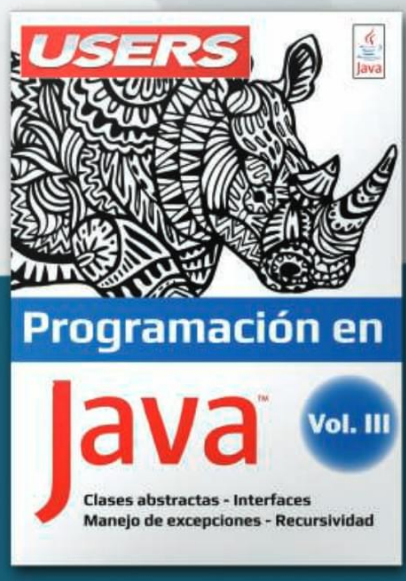
Clases abstractas - Interfaces
Manejo de excepciones - Recursividad

CURSO DE

PROGRAMACION EN JAVA







Aprende a programar aplicaciones robustas y confiables. Escribe tus códigos una vez y ejecútalos en cualquier dispositivo.

Programación en Java™

Vol. III

Clases abstractas - Interfaces
Manejo de excepciones - Recursividad

USERS

Título: Programación en Java III / **Autor**: Carlos Arroyo Díaz
Coordinador editorial: Miguel Lederkremer / **Edición**: Claudio Peña
Maquetado: Marina Mozzetti / **Colección**: USERS ebooks - LPCU291

Arroyo Díaz, Carlos
Programación en JAVA III : Clases abstractas. Interfaces. Manejo de excepciones. Recursividad / Carlos Arroyo Díaz. - 1a ed. - Ciudad Autónoma de Buenos Aires : Six Ediciones, 2019.
Libro digital, PDF - (Programación en JAVA)

Archivo Digital: descarga y online
ISBN 978-987-4958-11-2

1. Lenguaje de Programación. I. Título.
CDD 005.13

ACERCA DE ESTE CURSO

Java es un lenguaje de programación que sigue afianzándose como un estándar de la Web y, por eso, año tras año, aparece en el tope de las búsquedas laborales de programadores.

Es por esto que hemos creado este curso de **Programación en Java**, donde encontrarán todo lo necesario para iniciarse o profundizar sus conocimientos en este lenguaje de programación.

El curso está organizado en cuatro volúmenes, orientados tanto a quien recién se inicia en este lenguaje, como a quien ya está involucrado y enamorado de Java.

En el **primer volumen** se realiza una revisión de las características de este lenguaje, también se entregan las indicaciones para instalar el entorno de desarrollo y, posteriormente, se analizan los elementos básicos de la sintaxis y el uso básico de las estructuras de control.

En el **segundo volumen** se presentan las clases en Java, se realiza una introducción a los conceptos asociados a la Programación Orientada a Objetos y también se profundiza en el uso de la herencia, colaboración entre clases y polimorfismo.

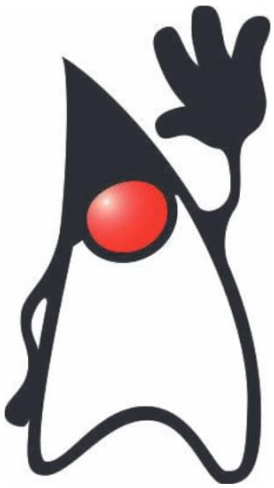
El **tercer volumen** contiene información sobre el uso de las clases abstractas e interfaces, el manejo de excepciones y la recursividad.

Finalmente, en el **cuarto volumen** se enseña el uso de las estructuras de datos dinámicas, el acceso a bases de datos y la programación Java para Android.

Sabemos que aprender todo lo necesario para programar en Java en tan solo cuatro volúmenes es un tremendo desafío, pero conforme vamos avanzando, el camino se va allanando y las ideas se tornan más claras.

¡Suerte en el aprendizaje!

SUMARIO DEL **VOLUMEN III**



PRÓLOGO

Java es un lenguaje maduro y robusto que, desde su nacimiento en el año 1995, ha demostrado que vino para quedarse y ha logrado evolucionar hasta convertirse en el lenguaje más utilizado en el mundo tecnológico.

Durante nuestro aprendizaje de cualquier lenguaje de programación podemos encontrarnos con variados libros, pero solo a través de la experimentación y escudriñando las entrañas de la Web (foros, blogs, under sites), podremos obtener los conocimientos necesarios para enfrentar el aprendizaje de mejor forma.

Mientras recorremos este camino, contar con un curso como este es fundamental pues se presenta como un compendio de todo lo que necesitamos para enfrentar un lenguaje de programación, permitiéndonos lograr, a través de ejemplos concretos, un mejor aprendizaje de los distintos temas necesarios para programar en Java.

Desde el primer momento somos optimistas en que este curso será para ustedes un desafío muy significativo y, por otro lado, logrará convertirse en una excelente guía del lenguaje Java.

Carlos Arroyo Díaz

Acerca del autor

Carlos Arroyo Díaz es programador de profesión, escritor especializado en tecnologías y docente por vocación. Se desempeña desde hace más de 20 años como docente en Informática General y en los últimos 10 años enseña programación en Java y Desarrollo Web. También se ha desempeñado como mentor docente en el área de Programación en varios proyectos del Ministerio de Educación de la Ciudad Autónoma de Buenos Aires, enseñando a programar a grupos de jóvenes.

Clases abstractas e interfaces

En este capítulo hablaremos de las clases abstractas, aprenderemos a implementarlas y veremos cuándo es conveniente usarlas. También conoceremos la herencia múltiple y tocaremos temas como el desacoplamiento de las clases, la abstracción a través de ellas y el uso de la interfaz **Comparable**.

01

CLASES ABSTRACTAS

Cuando pensamos en una clase, asumimos que vamos a crear objetos de ese tipo; no obstante, en ciertos casos es conveniente declarar clases para las que nunca crearemos instancias de esos objetos, a este tipo de clases se las conoce como **clases abstractas**. Al ser utilizadas en las jerarquías de herencia, las llamaremos como **superclases abstractas**, pues solo son utilizadas en superclases y nunca pueden instanciar objetos; servirán para proporcionar una superclase apropiada a partir de la cual heredan otras clases.

Vayamos al ejemplo de la clase **Empleado**. Si creamos una clase que esté por debajo de la clase **Jefe**, por ejemplo **Gerente**, a simple vista pareciera que **Gerente** es una clase más. Sin embargo, por jerarquía de clases (**niveles de herencia**), al estar por debajo de las otras dos, pasará a tener una especialización ya que heredará los métodos y los atributos de **Jefe**, y esta a su vez hereda los de **Empleado**. Entonces, tener más métodos y atributos supone que es más potente que las otras dos.

Si nos movemos al revés (de abajo hacia arriba), nos daremos cuenta de que las clases se vuelven más genéricas y, cuando llegamos al nivel superior, las clases se vuelven más abstractas.

Figura 1. Vista de la jerarquía de clase y de la regla “es un” que nos asegurará una herencia sin errores.

Vayamos más lejos aún: si creamos esta vez una clase que esté por encima de **Empleado**, por ejemplo la clase **Persona**, al estar jerárquicamente por encima de todas, tiene menos métodos que cualquiera que la suceda.

Siguiendo la regla **"es un..."**, un empleado es una persona, pero una persona no siempre es un empleado.

De la misma forma sucede cuando implementamos métodos en las distintas clases, por nombrar **unSueldo()** y **unNombre()**. En la clase (imaginaria) **Persona**, solo podríamos utilizar el método **unNombre()** puesto que no siempre una persona tiene un sueldo. El método **unSueldo()** sí quedaría en la clase **Empleado**, de tal forma que esta puede heredar a las otras dos; los jefes y los gerentes ganan sueldos.

La idea de esta nueva implementación es que, cuando utilicemos este concepto, tratemos en lo posible de elevar el nivel de los métodos y atributos más generales, y especialicemos a las clases que no llevan este concepto.

Si creamos una clase **Pasante** (que no cobra ningún sueldo), esta va a heredar los atributos y los métodos de la clase **Persona** y no de **Empleado**, y con esto seguimos respetando el diseño de la herencia ("es un...").

¿Qué sucederá si queremos crear un método que se llame **unaDescripcion()**, es decir, que traiga la descripción de los empleados? Al ser un método más bien general, tendemos que optar por colocarlo más arriba (en la clase **Persona**); sin embargo, dentro de este método podríamos poner cosas como, por ejemplo, el ID del empleado y, si nos remitimos a la clase **Pasante**, este no va tener un ID (recordemos que un pasante no es un empleado). En otras palabras, el método no siempre va a ser igual para todas las clases.

En estos casos, nos veremos obligados a crear los llamados **métodos abstractos**. La sintaxis de estos métodos es:

```
public abstract [tipoDato] [nombreMetodo]();
```

Como podemos observar no lleva un bloque con las llaves (**{}**), porque simplemente se definen.

Una regla de Java es que, cuando declaramos un método abstracto, estamos obligados a que la clase donde este convive debe ser necesariamente declarada como abstracta. La sintaxis sería la siguiente:

```
abstract class [nombreClase]{
…
}
```

Además de esto, hay que resaltar que, de esta forma, estamos siguiendo un patrón en el diseño de la jerarquía de la herencia, lo que nos lleva a otra regla: que todas las clases que estén por debajo de la clase que acabamos de abstraer van a estar obligadas a **sobrescribir** ese método abstracto o los que tenga.

Si vamos a nuestro código, trabajaremos en una clase nueva en la que implementaremos la clase abstracta **Persona**. Reutilizaremos el código de los ejemplos anteriores y para ello vamos a eliminar todo lo concerniente a los datos del empleado (nombre) y solo nos quedaremos con lo siguiente:

```
class Empleado{
    //constructor de clase
      public Empleado(String nom, double sue, int año,int mes, int
dia){
        sueldo = sue;
        GregorianCalendar calendario = new GregorianCalendar(año,
mes-1, dia);
        altaIngreso = calendario.getTime();
        }

      public  double unSueldo(){
          return sueldo;
      }
      public Date unaFechaAlta(){
          return altaIngreso;
      }
      //SETTER
      public void aumentaSueldo(double porcentaje){
         double aumento = (sueldo * porcentaje)/100;
         sueldo += aumento;
      }
```

```
        // variables de clase
        private double sueldo;
        private Date altaIngreso;
    }
```

Ahora crearemos la siguiente clase:

```
class Persona{
   public Persona(String nom){ //constructor de la clase
            nombre = nom;
        }

        private String nombre;
  }
```

Observamos que, por el momento, solo le pusimos el modificador **public**. Crearemos los métodos que vimos en la parte teórica:

```
public String unNombre(){ //Getter
        return nombre;
    }
```

Este es un **getter** que va a retornar el nombre. Veamos cómo se implementa un método abstracto:

```
public abstract String unaDescripcion();
```

Errores en la abstracción de clases

Cuando trabajamos con abstracción de clases, generalmente se cometen errores típicos que en las clases comunes no sucederían; uno de ellos es tratar de instanciar objetos, en consecuencia, nos llevará a un error de compilación. Otro error es que, si no se implementan métodos abstractos de la superclase abstracta en una subclase (que no sea abstracta), también se producirá un error en la compilación del programa; para ello debemos sobrescribir estos métodos, de la siguiente forma: **@Override public void nombreMetodoAbstracto(){...}**

El programa nos indicará que no reconoce este método; por otro lado, en la clase **Persona** también nos da un error, y el IDE nos recomienda que pongamos a la clase como abstracta. Corregimos de la siguiente forma:

```
abstract class Persona{
}
```

Ahora, en la clase **Empleado**, vamos a heredar de la clase que acabamos de construir:

```
class Empleado extends Persona{
    public Empleado(String nom, double sue, int año,int mes, int
dia){
            super(nom);
}
```

En el código anterior hemos colocado la palabra reservada **super** y, dentro de ella, el parámetro que hereda de la superclase (**nom**).

Lógicamente, empezará a presentar errores porque nos falta sobrescribir el método abstracto. Lo hacemos de la siguiente forma:

```
        @Override
        public String unaDescripcion(){
//sobreescitura del método
            return "Este empleado tiene un sueldo de: " +  +sueldo;
        }
```

Siguiendo el ejemplo, construiremos la clase **Pasante**, que va a heredar también de la clase **Persona**:

```
class Pasante extends Persona{

        public Pasante(String nom, String carre) {
            super(nom);
            carrera = carre;
        }
```

```
        private String carrera;
    }
```

La nueva clase **Pasante** va a tener las características heredadas de **Persona**, pero también las propias (**carrera**). Sabemos que el IDE nos sigue presentando errores, porque aún no hemos sobrescrito el método abstracto **unaDescripcion()**:

```
  @Override
      public String unaDescripcion(){
          return "Este alumno estudia la carrera de: "+
carrera;
      }
```

Una vez terminadas con las implementaciones y correcciones vamos dentro de la clase **main** para implementar los objetos que usaremos:

```
  public static void main(String[] args) {
         Persona[] persona = new Persona[2];
         persona[0]=new Empleado("Mateo Velarde",
25000,2008,03,22);
         persona[1]=new Pasante("Daniela Strauss", "Cont-
abilidad");

          for(Persona p: persona){
              System.out.println(p.unNombre()+
p.unaDescripcion());
          }
    }
```

En el código anterior verificamos que hemos creado un array para el objeto **persona** y que hemos pasado dos elementos **[2]**. Luego instanciamos dos objetos: uno para un empleado y otro para un pasante. Cuando ejecutamos el programa mostrará lo siguiente:

```
Mateo Velarde Este empleado tiene un sueldo de: 25000.0
Daniela Strauss Este alumno estudia la carrera de: Contabilidad
```

Casting en Java

Hacer la selección de los elementos constitutivos del lenguaje Java es casi una parada obligatoria. Este tema, que por el momento no lo habíamos tocado por cuestiones técnicas de falta de espacio no es menos importante a la hora de trabajar con los tipos de datos y objetos.

El **casting** en Java consiste en decirle al compilador que un objeto de tipo A es, en realidad, de tipo B más específico. No se está realizando ningún tipo de magia o conversión al ejecutar casting, estamos esencialmente diciendo al compilador "confía en mí, sé lo que estoy haciendo y puedo asegurarte que este objeto es en realidad un objeto". Por ejemplo:

```
Object elem = "cadena";
String cadena = (String)elem;
```

El objeto que se almacena en **elem** es en realidad una cadena y, por lo tanto, podemos remitir a una cadena sin ningún problema.

Hay dos maneras en que esto podría salir mal. En primer lugar, si estamos compartiendo dos tipos en jerarquías de herencia completamente diferentes, entonces el compilador sabrá que estamos haciendo algo indebido y por ende, fallará:

```
String elem = "cadena";
Integer cadena = (Integer)elem;
//Aquí es donde falla la compilación
```

En segundo lugar, si están en la misma jerarquía, pero siguen siendo inválidos, se lanzará una **ClassCastException** en tiempo de ejecución:

```
Number elem = new Integer(5);
Double n = (Double)elem;
//ClassCastException thrown here
```

Esto significa que se ha violado la confianza del compilador. Le dijimos al compilador que puede garantizar que el objeto es de un tipo particular, y no lo es.

¿Por qué necesita casting? Bueno, para empezar, solo lo necesita cuando va de un tipo más general a un tipo más específico. Por ejemplo, **Integer** hereda de **Number**, así que si lo que queremos es almacenar un entero como un número, entonces está bien (ya que todos los números enteros son números). Sin embargo, si queremos hacerlo a la inversa, vamos a necesitar seleccionar, porque no todos los números son **Integer**, tenemos también **Double**, **Float**, **Long**, **Byte**. Incluso si solo hay una subclase en el proyecto o el JDK, podríamos crear fácilmente otro y distribuirlo, por lo que no existe ninguna garantía.

En cuanto al uso para el casting, todavía se ve que en algunas bibliotecas anteriores a Java 5 se utilizó en gran medida en colecciones y varias otras clases, ya que todas las colecciones han trabajado en la adición de objetos y luego arrojaron el resultado que obtuvo de nuevo la colección. Sin embargo, con la incorporación de los **genéricos**, ha desaparecido gran parte del uso de esta colección, porque estos proporcionan una alternativa mucho más segura, sin el peligro que ocasiona la **ClassCastExceptions** (de hecho, si se utilizan genéricos de forma limpia y se compila sin advertencias, tendremos la garantía de que no obtendremos un **ClassCastException**).

Para seguir con los ejemplos veamos cómo podemos implementar el **casting**:

```
  misEmpleados[5] = new Jefe("José López",55000,2001,10,27);
//casting
   Jefe jefeSistemas = (Jefe)misEmpleados[5];
      jefeSistemas.ganaIncentivo(6000);
```

Donde instanciamos a un nuevo jefe, sin embargo, hemos aplicado el **casting**, ya que sabemos que, si bien es un jefe, lo estamos instanciando como un empleado y, para evitar engañar a la máquina virtual. Sin embargo, si hacemos esto:

```
  Jefe jefeSistemas = (Jefe) misEmpleados[1];
```

Nos va a dar error de excepción de casting –**ClassCastException**– como se explicaba en la introducción del tema. La razón es que el índice **[1]** ya está asignado para un empleado y no para un jefe.

Clase y método final

Ya hemos trabajado con la palabra clave **final**, pero esta vez lo haremos con referencia a clases y métodos.

Cuando una clase es **final**, implica que se va a detener la cadena de la herencia, para ello nos apoyaremos en el concepto de la **jerarquía de clases**. En nuestro ejemplo es bastante claro que existe una jerarquía, ya que la clase **Empleado** es la clase de la cual se heredan algunas características a la clase **Jefe**.

Si creáramos una clase **Director** dentro de la clase **Jefe** (que herede de ella), se notaría aún más el concepto de jerarquías ya que, a la vez que hereda de la clase **Jefe** algunos métodos y tal vez atributos, por jerarquía también lo hará de la clase **Empleado**. De esta forma, se podrían crear más clases por debajo de **Director**, es decir, no hay limitaciones para jerarquizar clases en Java.

Sin embargo, existen límites que los programadores podríamos poner, para que luego de una clase no se extiendan más clases.

Siguiendo el ejemplo, deberíamos declarar la clase **Jefe** como **final**, de tal forma que ponga un límite allí y se rompa la cadena de extensiones de más clases que, a futuro, hereden de **Jefe**.

Veamos en el ejemplo que tenemos en el proyecto. Para ello vamos a crear una nueva clase llamada **Director**, y esta a su vez heredará de la clase **Jefe**:

```
public class Director extends Jefe {
    ...
}
```

Jerarquía en Java

Antes que sepamos lo que es una jerarquía en Java, es importante diferenciar entre los conceptos de herencia y de package. Un **package** es una agrupación arbitraria de clases, una forma de organizar las clases, en tanto, que la **herencia** consiste en crear nuevas clases sobre la base de otras ya existentes. Las clases incluidas en un package no derivan, por lo general, de una única clase. La **jerarquía en clases** consiste tener dentro de un esquema en el que la clase padre tenga características en común, y cada clase hija, las específicas.

Sabemos que, al principio, nos va a marcar un error, que más bien es conceptual, puesto que no tenemos el **constructor por defecto**.

Entonces utilizaremos la ayuda del IDE. Hacemos **clic derecho** dentro de la clase creada, elegimos el comando **Insertar código** y, luego, procedemos con la opción **Constructor...**

Lo que nos dará:

```
public Director(String nom, double sue, int año, int mes, int dia) {
        super(nom, sue, año, mes, dia);
    }
```

Nos trae los parámetros de la clase padre (**Jefe**). Si procedemos a extender otra clase a la clase **Director**, por concepto sabemos que Java lo va a permitir. En este caso, le vamos a colocar delante de la clase **Jefe** la palabra **final** y veamos qué sucederá:

```
final class Jefe extends Empleado{
...
}
```

Automáticamente el programa empezará a dar errores, para indicarnos que la clase **Director** no puede heredar de la clase **Jefe**, y este era nuestro objetivo: no permitir que ya nadie realice herencia de esta clase **Jefe**.

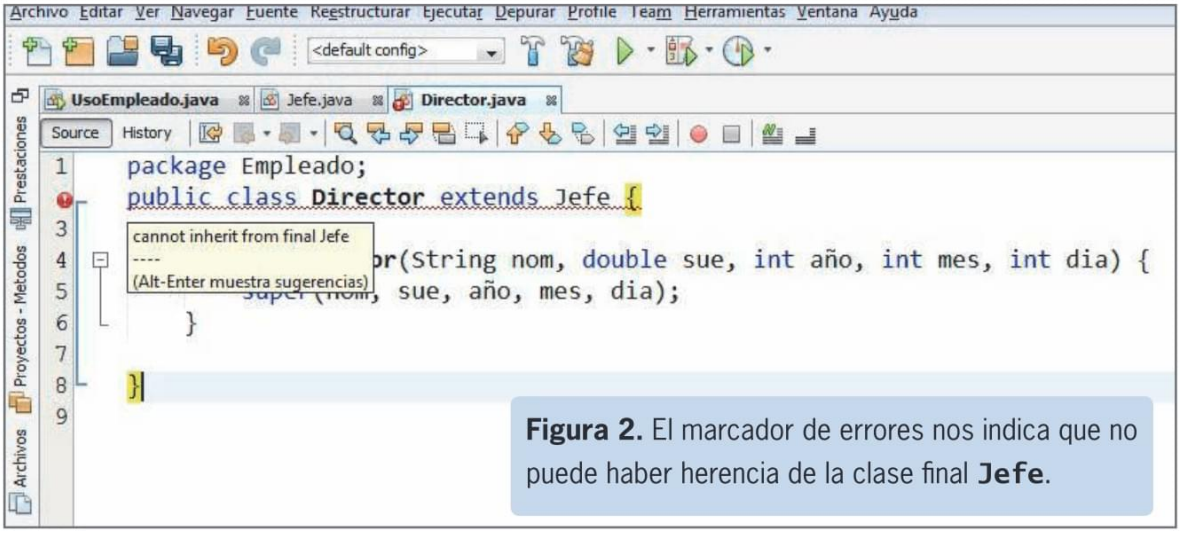


Figura 2. El marcador de errores nos indica que no puede haber herencia de la clase final **Jefe**.

Ahora veamos qué pasa cuando colocamos **final** delante de un método. Recordemos que habíamos tratado con la sobrescritura de métodos, con la que nos quedamos maravillados por los efectos que producía esta acción en nuestro programa. Sin embargo, no siempre este tipo de actividades nos resultarán útiles y, de hecho, puede que molesten a determinadas acciones. Por eso, tal vez necesitemos ponerles límites a ciertos métodos, para que no puedan ser sobrescritos. Veamos en la clase **Empleado**:

```
public final double unSueldo(){
          return sueldo;
      }
```

Colocamos un **final** y, como era de esperarse, la clase heredada **Jefe** va a sufrir las consecuencias:

```
@Override
 public double unSueldo(){
 ...
 }
```

El marcador de errores nos va a indicar que no podemos sobrescribir este método. Una solución a este percance es ponerle otro nombre a dicho método:

```
//@Override
 public double unSueldoJefe(){
 ...
 }
```

Nótese que tuvimos que comentar **@Override** puesto que antes lo colocamos por sugerencia del IDE para indicar que era un método sobrescrito.

En la comunidad de programadores de Java, existen programadores que apoyan el proyecto y, con frecuencia, mejoran y actualizan la API. También es frecuente que varias de las clases o los métodos declaren los métodos de manera final. Un ejemplo que vimos con Gregorian Calendar es el método **getTime()**. Si nos remitimos a la página oficial

de la API de Java, veremos que este método tiene declarada la palabra **final** adelante; entonces si quisiéramos crear un método que heredara de Calendar, no podríamos sobrescribir el método **getTime**. De la misma forma ocurre con clases de la API que estén declaradas con **final**, un ejemplo que podemos usar es la clase **String**.

Modificadores de acceso protected

Verifiquemos cómo se comportan los distintos modificadores cuando se los llama desde cualquier parte del programa; con muchos de ellos ya estuvimos trabajando a lo largo de estos ebook.

TABLA DE ACCESIBILIDAD

MODIFICADOR	CLASE	PAQUETE	SUBCLASE	TODOS
Public	**Sí**	**Sí**	**Sí**	**Sí**
Protected	**Sí**	**Sí**	**Sí**	**No**
Private	**Sí**	**No**	**No**	**No**
Por defecto	**Sí**	**Sí**	**No**	**No**

Figura 3. Tabla de accesibilidad de los modificadores de acceso.

Esta vez hablaremos del modificador **protected**, que según la tabla es accesible desde la misma clase; desde otra clase (siempre que esté en el mismo paquete); desde una subclase, aunque esté en otro paquete, pero no es accesible desde otra clase normal que esté en otro paquete.

Para que esto nos resulte más fácil de entender, mostraremos un ejemplo, que tendrá la siguiente estructura:

```
pkgprotected
        Clase1.java
        Clase2.java (Esta llevará el método main).
pkgPruebas
        Clase3.java
```

En la **Clase1** escribiremos el siguiente código:

```
  package pkgprotected;
  public class Clase1 {
//modificador por defecto
      int var1 = 5;
//modificador por defecto
      int var2 = 6;

//acá debería ir el constructor por defecto

      String miMetodo(){
          return "El valor de var2 es: " + var2;
      }

  }
```

Hemos declarado dos variables, ambas con **modificadores por defecto** (no se escribe nada). Luego un método **String** que retorna un texto con una de las variables.

Recordemos que, cuando creamos una clase, debemos crear también un constructor de la clase; si omitimos este proceso, Java asumirá que le pondremos un **constructor por defecto**. Cuando instanciamos en la **Clase2**:

```
  public class Clase2 {
      public static void main(String[] args) {
          Clase1 objeto1 = new Clase1();
//instancia de objeto
          objeto1.
//(acá deberían aparecer las variables y los métodos de la Clase1)

      }
  }
```

entonces sucede que al escribir **objeto1** el IDE nos deja ver las variables y el método de la **Clase1**; esto es porque ellas no tienen modificador. Si nos fijamos en la tabla serán visibles.

En cambio, si a las variables les anteponemos el modificador **private** (encapsulamos), desde la **Clase2** ya no podremos verlas.

Si vamos un poco más a fondo en este tema del acceso a los miembros, llegamos a la **Clase3**, que se encuentra en otro paquete (**pkgPruebas**). Si queremos hacer que herede de la **Clase1**:

```
package pkgPruebas;
public class Clase3 extends Clase1{

}
```

El programa nos marcará que no reconoce **Clase1**, esto es porque está en otro paquete (**pkgprotected**). Para solucionar esto, debemos importar el paquete:

```
import pkgprotected.Clase1;
```

Ahora vayamos a la **Clase2** e instanciemos un nuevo objeto:

```
Clase3 objeto2 = new Clase3();
```

Nuevamente marcará un error, porque la **Clase3** está en otro paquete, entonces debemos importar su paquete:

```
import pkgPruebas.Clase3;
```

Con esto, en la **Clase2** tenemos instancias de la **Clase1** y de la **Clase3**. Parece sencillo, pero, si no lo prevemos, de verdad nuestro programa se convertirá en un laberinto difícil de sortear.

Desde el **objeto2** intentaremos acceder a las variables de la **Clase1**:

```
objeto2.
//aquí esperamos que el IDE proponga
```

Como podemos comprobar, no hay manera de verlas; esto se debe a que los modificadores no están como **private** o están por defecto (debemos remitirnos a la tabla de modificadores de acceso).

Entonces, aquí pondremos en escena al modificador **protected**:

```
    protected int var1 = 5;
    protected int var2 = 6;

    protected String miMetodo(){
...
}
```

En cuanto coloquemos este modificador de acceso, ya tendremos visibilidad de las dos variables y del método. El modificador **protected** cumple una función parecida al de **private**, pero es más permisivo, en especial cuando implementamos herencia.

Clase Object

La clase **Object** es llamada por algunos autores la clase cósmica, puesto que está por encima de todas las existentes. Esto quiere decir que, cuando construimos una clase en nuestros programas, estas van a heredar de la clase **Object**. Asimismo, las clases de la API de Java también heredan de ella.

Paquetes predeterminados

En Java existe un paquete predeterminado que se llama **java.lang**; cuando queremos utilizar elementos de un paquete y este no es el paquete por defecto, hay que importarlo. Para usar un paquete en una de nuestras clases, antes de definir la clase debemos introducir la instrucción **import**. Por ejemplo, si queremos usar el paquete **java.awt**, hay que introducir, entre la línea del paquete de nuestro proyecto y la línea del nombre de la clase, la siguiente instrucción: **import java.awt.*;**

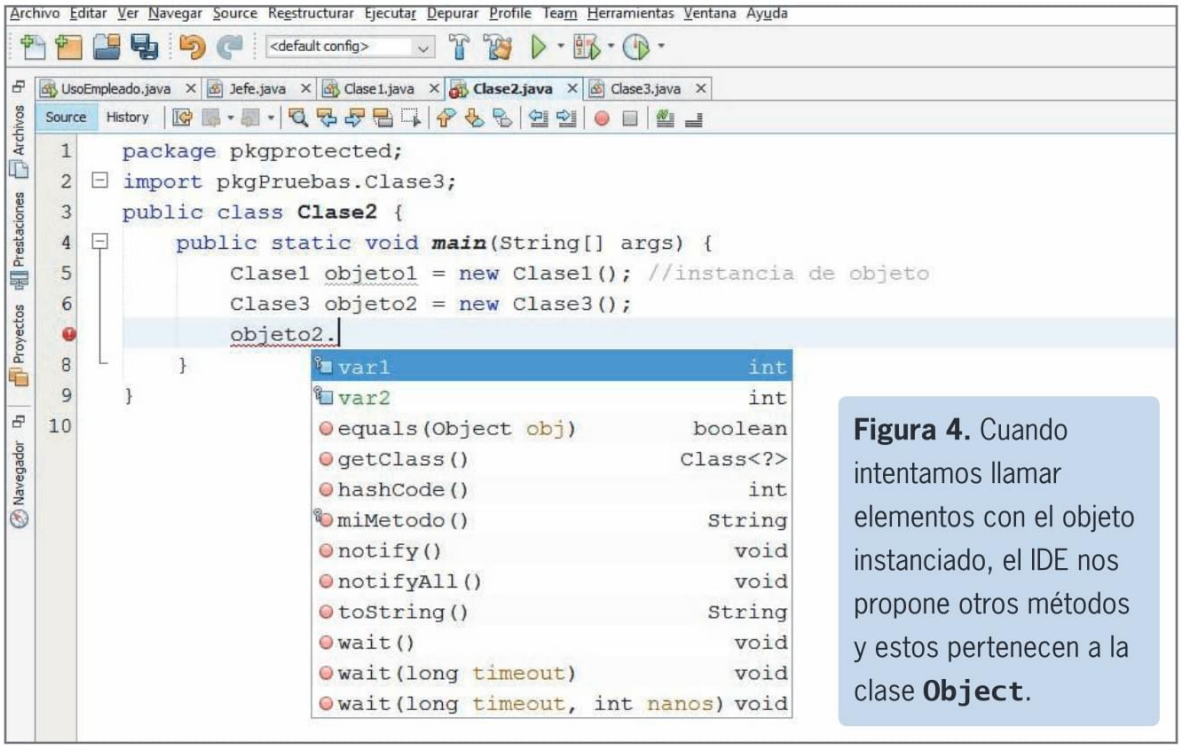


Figura 4. Cuando intentamos llamar elementos con el objeto instanciado, el IDE nos propone otros métodos y estos pertenecen a la clase **`Object`**.

CREACIÓN Y USO DE INTERFACES

Las **interfaces** son un conjunto de directivas o comportamientos que deben seguir las clases, es decir, definen y estandarizan la forma en que van a interactuar los objetos entre sí. Por ejemplo en un auto, de qué manera interactúa el conductor con el encendido, la caja de cambios, los pedales, etcétera. En este caso, el conductor sabe cómo actuar frente a estos elementos, es decir, conoce qué debe hacer para que funcionen, pero no sabe cómo funcionan (a menos que el conductor sea un mecánico).

Las principales características que las interfaces poseen se enumeran a continuación:

- Definen un **tipo de dato** en términos de los métodos públicos que debe ofrecer.
- Solo pueden contener métodos abstractos.
- Se almacenan en un archivo **.class**.

- No se pueden instanciar (no usan **new**).
- Todos sus métodos son **public** y **abstract**. No se implementan.
- No tienen variables, en vez de ellas van a tener constantes.
- No son clases, aunque puedan entenderse como una "clase abstracta".

Esto nos lleva a la siguiente pregunta: si se parecen tanto a las clases abstractas, ¿qué nos lleva a crear las interfaces?

La respuesta es que Java no permite la **herencia múltiple**, y además la comprobación estricta de tipos que hace permanentemente Java.

¿Qué pasaría si a la clase **Jefe** quisiéramos implementarle dos métodos obligatorios para esta clase? No podemos utilizar una clase abstracta, pues esta ya está heredando de la clase **Empleado**. Esto es así, porque no podemos crear una clase abstracta en la que se definan estos métodos. Lo que se permite es implementar interfaces. La clase **Jefe** puede heredar de una o de varias interfaces. La forma de escribirlo es la siguiente:

```
public class Jefe extends Empleado implements Interfaz1, Interfaz2{
    ...
}
```

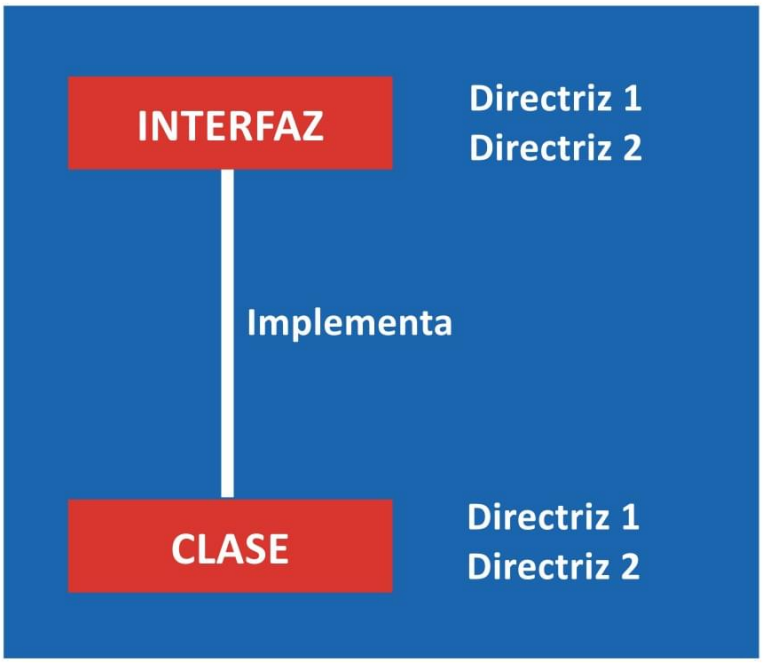


Figura 5. Gráfico del funcionamiento de las interfaces en Java y la interacción con las clases que intentan comunicarse.

La palabra clave **implements** sirve para indicarle a la clase **Jefe** que va a "heredar" de las interfaces 1 y 2; esta es una forma de saltar las reglas de la no permisión de la herencia múltiple.

Para continuar con la temática presentada trabajaremos con el proyecto del **Empleado**.

Recordemos que la última salida que tuvimos de este paquete fue la siguiente:

```
Nombre: Juan Pérez Sueldo: 15900.0 Fecha Ingreso: Sat Mar 18 00:00:00 GMT-03:00 2000
Nombre: Marta Rodríguez Sueldo: 12720.0 Fecha Ingreso: Thu May 22 00:00:00 GMT-03:00 2003
Nombre: Martín de Robertis Sueldo: 16960.0 Fecha Ingreso: Fri Oct 30 00:00:00 GMT-03:00 1998
...
```

Interfaz Comparable

La interfaz **Comparable** es una manera de implementar el uso de interfaces, pero esta vez utilizando una predefinida, es decir, que ya hemos importado o una que se incluye en la API de Java.

Pongamos una consigna en el ejemplo de los empleados, ya que la forma en que salen los resultados no establece un orden específico. Vamos a suponer que queremos ordenar por sueldo. Como estamos trabajando con arrays, recordemos que para ordenar este tipo de entidades utilizaremos un método llamado **sort**. Como buena práctica recomendamos siempre visitar la página oficial de Oracle (**https://docs.oracle.com/javase/9/docs/api/**), en donde se encuentra la API de Java, y verificar cómo se implementa cada uno de los métodos que vamos nombrando.

Si investigamos un poco dentro de la API de Java (9), encontraremos la clase **Arrays** y, dentro de ella, muchos métodos ordenados alfabéticamente; cuando hallemos **sort**, este apuntará a distintos tipos (**char**, **int**, etcétera). Como en el ejemplo **Empleado** es un objeto, debemos buscar el **sort** que apunte a los objetos: **sort(Object[] a)**; hacemos clic en este enlace para adentrarnos aún más en esta web. Si traducimos lo que nos describe este elemento entendemos que ubica los elementos en orden natural (de mayor a menor o alfabéticamente), pero lo que llama muchísimo la atención es esta frase: todos los elementos de un array que utilicen **sort** deben implementar la interfaz **Comparable**.

Esto quiere decir que, si queremos utilizar este métodos, debemos implementar la interfaz **Comparable**, puesto que todos los elementos de la clase **Empleado** son de este tipo de objeto.

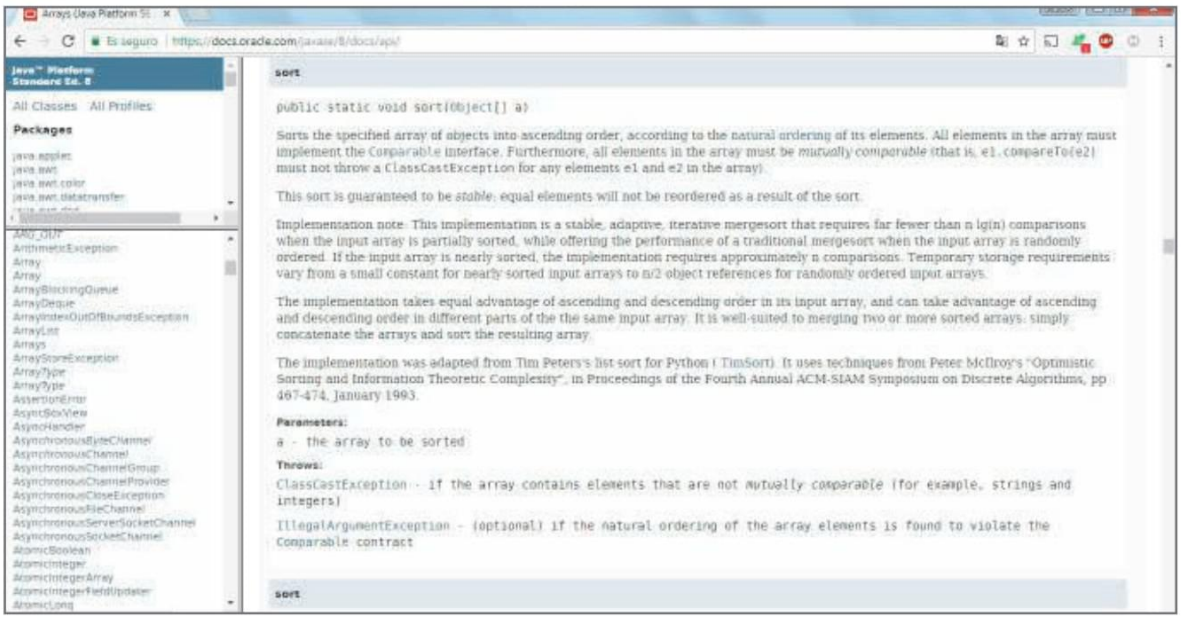


Figura 6. En la API de Java, la clase **Arrays** contiene el método **sort**. En **Object** notemos la palabra clave **Comparable**, vital para el concepto de interfaces.

Si nos metemos en el enlace **Comparable**, verificamos todas las opciones que podemos implementar con esta interfaz, pero tiene un único método, que describe lo siguiente:

```
int compareTo(T o)
```

Donde **T** es, en nuestro ejemplo, un objeto de tipo **Empleado**.

Traducido sería que este método compara objetos en un orden específico; para nuestro ejemplo compararía a los empleados entre sí.

En nuestro código, iremos a la clase principal y después del **foreach** escribiremos:

```
...
for(Empleado e: misEmpleados){
            e.aumentaSueldo(6);
        }
        Arrays.sort(misEmpleados);
```

Como el programa no demuestra ningún error, nos tienta a pulsar **play** o **F6**:

```
  Exception in thread "main" java.lang.ClassCastException: Empleado.
Empleado cannot be cast to java.lang.Comparable
```

¡Tamaño error! y por cierto desconocido hasta ahora, pero si nos detenemos a leerlo cuidadosamente aparece una palabra que ya estuvimos investigando, **Comparable**, es decir que **Empleado** no puede hacer un cast de la interfaz **Comparable**.

Deberíamos ir inmediatamente a la clase **Empleado** y escribir:

```
class Empleado implements Comparable{
...
```

Pero surge que hay ciertas reglas, una de ellas dice que estamos obligados a escribir el método de la interfaz, en este caso de **Comparable**. Si nos acercamos al marcador de errores, este nos va a indicar que debemos implementar el método abstracto **CompareTo** y, además, que **Empleado** no es una clase abstracta.

Entonces debemos sobrescribir este método de la siguiente manera: dentro de la clase **Empleado**, nos vamos a posicionar al final, antes de los campos de clase:

```
@Override
      public int compareTo(Object objeto){
          Empleado unEmpleado = (Empleado) objeto;
          if(this.sueldo<unEmpleado.sueldo){
              return -1;
          }
           if(this.sueldo>unEmpleado.sueldo){
              return 1;
      }
           return 0;
      }
```

Desglosando el código, creamos el método **compareTo()** y le damos por parámetro **Objeto** con el nombre **objeto**.

Luego hacemos un casting del objeto **Empleado** a una variable objeto (**unEmpleado**), así podemos comparar el sueldo de los empleados.

Por último, creamos una secuencia de **if** para que pueda comparar los sueldos. La API de Java nos decía que, si A es mayor con respecto a B, nos devuelve **1**; si B es mayor que A, devuelve **-1**; cuando ambos sueldos sean iguales devuelve **0**.

Cuando ejecutemos el programa, efectivamente se habrá ordenado el sueldo de menor a mayor.

Con esto llevamos a la práctica el uso de interfaces, al establecer un comportamiento de la clase que la implementa.

Uso de instanceof

El operador **instanceof** solo puede usarse con variables que contengan la referencia a un objeto, es decir, variables que apuntarán a la dirección en la memoria en la que está almacenado el objeto. Es decir, una variable de referencia normal.

El objetivo del operador **instanceof** es conocer si un objeto es de un tipo determinado. Por **tipo**, nos referimos a una clase o a una interfaz. Para ello recordemos que el objeto debería cumplir con la regla "es un..." para esa clase o esa interfaz.

Vamos a ejemplificar este concepto y lo haremos incrementando las funcionalidades a nuestros códigos del proyecto Empleados, en la clase **Uso_Empleado**:

```
  Empleado jefeComercial = new Jefe("Salvador Ramírez", 40000, 2005,
14, 06);
      Comparable ejemploEmpleado = new Empleado("Daniela Rivas",
27000, 2010, 12, 30);
```

Instanciamos un nuevo jefe (comercial) con los datos y los argumentos.

Ya teníamos una interfaz **Comparable** (que creamos en el tema anterior), pero, como las interfaces no se pueden instanciar, utilizamos el **principio de sustitución** y creamos una instancia a la que llamamos **ejemploEmpleado** y le decimos que es del tipo **Empleado**.

Luego pasamos los parámetros correspondientes al constructor. Ahora utilizaremos el operador **instanceof** para comprobar si una instancia pertenece a una clase. Implementaremos un **if** para el siguiente código:

```
if(jefeComercial instanceof Empleado){
        System.out.println("Es de tipo Jefe");
    }
    if(ejemploEmpleado instanceof Comparable){
        System.out.println("Implementa la interface Comparable");
    }
```

Interfaces personalizadas

Podemos crear nuestras propias interfaces a fin de no depender de la API de Java. Para implementar una trabajaremos con las clases del proyecto Empleado y haremos el siguiente esquema:

Clases
Empleado
Jefe

Interfaces		
Empleados	->	recibeBonus()
Jefes	->	tomaDecisiones()

Aquí decidimos crear una interfaz **Jefes** con un método, puesto que queremos que todos los jefes tengan esa función.

Para crear una interfaz, debemos ir al proyecto que estamos trabajando y a la carpeta **Empleado**; hacemos clic con el **botón derecho**, luego elegimos la opción **nuevo**, y después la opción **Java Interface...**

En el cuadro para colocarle el nombre, le ponemos **Jefes**. Luego hacemos clic en el botón **Terminar**. Tendremos la siguiente línea de código:

```
public interface Jefes {

}
```

Por concepto sabemos que, si vamos a colocar métodos en nuestra interfaz creada, estos deben ser públicos y abstractos.

```
public abstract String tomaDecisiones(String decision);
```

Sin embargo, si omitimos esto, Java sobrentenderá que es así.

```
String tomaDecisiones(String decision);
```

Ahora debemos ir a la clase **Jefe** para implementar esta interfaz y decirle que herede de **Jefes**.

Pero nos encontramos ante una disyuntiva de concepto de Java, y es que no permite la herencia múltiple.

Aquí entran en escena las interfaces, que son una manera subjetiva de hacer herencias múltiples sin que atentemos contra la prohibición que mencionamos antes:

```
class Jefe extends Empleado implements Jefes{
...
}
```

La palabra clave **implements** es un análogo de **extends** en herencia.

Pero el programa nos marca un error; verificamos el marcador de errores del IDE y nos dice que la clase **Jefe** no es abstracta, y no va a sobrescribir el método que contenga **Jefes**.

Entonces implementaremos el método **tomaDecisiones**, lo haremos luego del constructor:

```
@Override
public String tomaDecisiones(String decision){
        return "Un miembro de la Gerencia tomó la decisión de: " +
decision;
    }
```

Como es un método sobrescrito, nos va a sugerir que le agreguemos **@Override** arriba de las instrucciones. Para poder corroborar que esto funcione, debemos ir a la clase principal (donde se encuentra el **main**) y, antes del **foreach**, escribiremos lo siguiente:

```
System.out.println(jefeSistemas.tomaDecisiones("Despedir al 5% del
personal de Sistemas"));
```

Cuando ejecutamos, verificamos que esto funciona.

```
Un miembro de la Gerencia tomó la decisión de: Despedir al 5% del personal de sistemas
```

Si instanciamos a un empleado: **misEmpleados[3]**.

Luego dejamos que el IDE nos dé las sugerencias, veremos que no aparecerá el método **tomaDecisiones**. Esto se debe a que quien implementa la interfaz es **Jefe**. Por lo tanto, solo un objeto instanciado de esa clase podrá ver el método mencionado.

Vamos a implementar la interfaz **Empleados**, que tiene un método (**recibeBonus**):

```
public interface Empleados {
    double recibeBonus(double aguinaldo);
}
```

Esta vez no hemos puesto lo que exige el programa, es decir, anteponerle que sea **public** y **abstract**. Dijimos que Java lo sobrentiende.

Crearemos un **bonus básico**, y haremos que sea constante, es decir, que sea fijo:

```
public static final double BONUSBASICO = 4000;
```

Ahora que ya tenemos las dos interfaces que habíamos fijado en el esquema inicial, les daremos una jerarquía al igual que hicimos con las clases.

```
public interface Jefes extends Empleados{
}
```

Inmediatamente nos va a marcar un error en la clase **Jefe** y, como la interfaz **Jefes** ahora hereda de **Empleados**, nos obliga a que la clase **Jefe** cree los métodos de la interfaz **jefes** como la de **Empleados**:

```
@Override
    public double recibeBonus(double aguinaldo){
    double prima = 2000;
    return Empleados.BONUSBASICO + aguinaldo + prima;
}
```

Acá le estamos creando una prima inicializada en 2000. Luego, cuando la instancia de la interfaz **Empleados** va a llamar a la constante **BONUSBASICO**, le sumamos un aguinaldo.

```
class Empleado implements Comparable, Empleados{
…}
```

Al implementar **Empleados** estamos obligados a implementar el método que este tenía:

```
@Override
      public double recibeBonus(double aguinaldo){
    return Empleados.BONUSBASICO + aguinaldo;
}
```

Retornamos el **BONUSBASICO** que recibe un jefe y el **aguinaldo**. Para ver los resultados escribiremos en el **main**:

```
System.out.println("El jefe de sistemas " +jefeSistemas.unNom-
bre()+ " tiene un bonus de: "
                  + jefeSistemas.recibeBonus(3000));
```

Cuando ejecutemos el programa, nos dará el siguiente resultado:

```
El jefe de Manuel Yáñez tiene un bonus de: 9000.0
```

Donde tienen **4000** de **BONUSBASICO**, **2000** de prima y **3000** de aguinaldo. Ahora, ¿qué pasa con los empleados?

```
System.out.println(misEmpleados[3].unNombre()+" tiene un bono de:
"+ misEmpleados[3].recibeBonus(1000));
```

Sin embargo, cuando esperamos que el IDE nos dé los métodos, no aparece **tomaDecisiones**, puesto que este no pertenece a los empleados, solo los jefes lo hacen. Cuando ejecutamos:

```
Juan Pérez tiene un bono de: 5000.0
```

Como vemos no tiene prima, puesto que solo la tienen los jefes.

DESACOPLAMIENTO DE CLASES

El **acoplamiento** de clases en la programación orientada a objetos implica la dependencia que existe entre dos clases. Esto significa que, si una clase (A) accede a elementos internos de otra clase (B), y esta clase (B) cambia algo en su estructura interna y es accedida desde la clase A, entonces la clase A se verá afectada por dichos cambios.

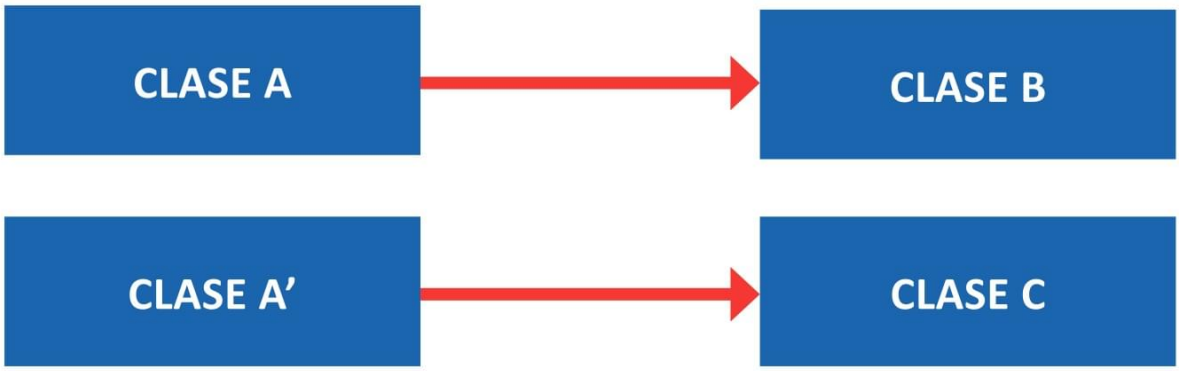


Figura 7. En la figura se muestra el acceso de la Clase A a la Clase B, y cómo una Clase A' (construida para este propósito) intenta acceder a una instancia de una Clase C.

Algo que reduce el acoplamiento entre las clases es la encapsulación. Entonces, si los elementos de la clase B van a permanecer ocultos o encapsulados, la clase A no los puede ver, por lo tanto, no se verá afectada. En este caso, estamos hablando de mayor desacoplamiento.

Si hay un mayor grado de desacoplamiento, entonces, esta Clase A solo accederá a métodos y atributos públicos de la Clase B.

Vamos a suponer que se desea generalizar la Clase A para que opere alternativamente sobre instancias de la Clase C. Dado que la Clase A está preparada para acceder solo a la Clase B, tendríamos, en principio, que construir una Clase A' con la lógica de la Clase A, pero preparada para referenciar objetos de la Clase C. Entonces, cada A y A' que se parezcan van a referenciar a la clase B y C, respectivamente. Esto podría llegar a solucionarse a través de métodos que ya estuvimos viendo, y para ello analizaremos un ejemplo.

Tenemos un programa **Timer** que debe operar en el encendido y apagado en determinadas horas de distintos sistemas: de un sistema de alarma, de un sistema de climatizador y de un grabador (de series o programas de TV).

Sistema de alarma:

```
  public class Timer {

//variable de instancia

      private Alarma dispositivo;
      boolean horaActivar, horaDesactivar;


//constructor de clase
      public Timer (Alarma d){
          dispositivo = d;
      }


//método que determina el encendido o apagado del dispositivo
      public void operar(){
          if(horaActivar)
              dispositivo.encender();
          else if(horaDesactivar)
              dispositivo.apagar();
      }

  }
```

La variable de instancia **dispositivo** de tipo **Alarma** es seteada en el constructor a través del parámetro **Alarma**.

Luego, el método **operar** actuará a través de una estructura de decisión invocando, según el caso, al método **encender** o **apagar** del dispositivo del sistema climatizador:

```
  public class Timer {
//variable de instancia
      private Climatizador dispositivo;
      boolean horaActivar, horaDesactivar;

//constructor de clase
      public Timer (Climatizador d){
          dispositivo = d;
      }
      public void operar(){
          if(horaActivar)
              dispositivo.encender();
          else if(horaDesactivar)
              dispositivo.apagar();
      }
  }
```

El código es similar al anterior, cambia el tipo que es **Climatizador** y el parámetro que lleva el mismo nombre.

Sistema grabador:

```
  public class Timer {
//variable de instancia
      private Grabador dispositivo;
      boolean horaActivar, horaDesactivar;

//constructor de clase
      public Timer (Grabador d){
          dispositivo = d;
      }
```

```
    public void operar(){
        if(horaActivar)
            dispositivo.encender();
        else if(horaDesactivar)
            dispositivo.apagar();
    }
}
```

De la misma manera, este código es similar al anterior; solo cambiará el tipo que es **Grabador** y el parámetro que lleva el mismo nombre.

En conclusión, viendo que la lógica de la programación del **Timer** es similar para cada uno de los sistemas tratados y que estos sistemas poseen los métodos para ser encendidos o apagados por este **Timer**, podríamos implementar una clase **Timer** única que sea capaz de controlar cualquiera de estos dispositivos.

Y esto es posible gracias a las abstracciones a través de las interfaces.

COHESIÓN

Usamos cohesión cuando indicamos cuán individual es una clase. Mientras más enfocada esté una clase, su cohesión será más alta. Esto es el objetivo de una buena práctica pues conlleva a un fácil mantenimiento y reutilización de las clases.

Supongamos que tenemos una clase que realiza los siguientes métodos:

```
class ReporteIngresos{
    void conectarDB(){ }
    void generarReporte() { }
    void guardarArchivo() { }
    void imprimir(){ }
}
```

Como ejemplo de cohesión quedaría de la siguiente forma:

```
class ReporteIngresos{
    Opciones getReporteOpciones(){…}
    void generarReporteIngresos (Opciones o){…}
}

class ConectarDB(){
    DBConectar getDBConectar(){…}
}

class Imprimir{
    ImprimirOpciones getImprimirOpciones(){…}
}

class GuardarArchivos{
    GuardarOpciones getGuardarOpciones(){…}
}
```

En lugar de tener una clase que haga todo, logramos cuatro clases específicas, más específicas o cohesivas.

RESUMEN CAPÍTULO 01

En este capítulo analizamos la abstracción de clases y aprendimos las características de esta metodología en la POO. Vimos los errores que ocurren con frecuencia en la abstracción y aprendimos para qué podemos utilizar el casting en Java. Más adelante, analizamos los métodos final, vimos para qué sirven y detallamos cuándo es conveniente usarlos. Conocimos la clase Object y también vimos las interfaces, analizamos sus características y sus formas de uso. Para terminar, conocimos la interfaz Comparable y el desacoplamiento de clases.

Actividades 01

Test de Autoevaluación

1. ¿Qué es la abstracción de clases? ¿Qué nos lleva a implementar este tipo de metodología en la POO?

2. ¿Qué tipo de errores ocurren con frecuencia en la abstracción?

3. ¿Para qué usamos el casting en Java?

4. ¿Qué son los métodos final? ¿Para qué sirven y cuándo es conveniente usarlos?

5. ¿Cuál es el motivo por el que anteponemos a una clase la palabra clave "final"?

6. ¿Qué es la clase Object?

7. ¿Qué es una interfaz?

8. ¿Cuáles son las características que diferencian a una interfaz de una clase?

9. ¿Qué es la interfaz Comparable?

10. ¿Qué es el desacoplamiento de clases?

Ejercicios prácticos

1. Cree un sistema de clases para la automatización de una casa (casa inteligente). Realice la abstracción de una clase.

2. En el ejemplo anterior genere el uso de interfaces, de tal forma que evite el acoplamiento entre clases.

3. Cree dos paquetes y, en ellos, deberá crear 2 y 3 clases, respectivamente. Mediante el uso de los modificadores de acceso, utilice protected y anote los resultados. Verifique con el uso de private.

Excepciones

En este capítulo abordaremos uno de los temas más difíciles de la programación en cualquier lenguaje: el manejo de las excepciones. Aprenderemos de qué se tratan, cómo y cuándo las vamos a utilizar, veremos el uso correcto del bloque try. Luego analizaremos el lanzamiento de excepciones a través de throws, el uso de los bloques catch, y cómo se especifican los manejadores de excepciones y la liberación de recursos a través del finally.

02

¿QUÉ SON LAS EXCEPCIONES?

Una **excepción** constituye una manera de indicar que existe un problema durante la ejecución del programa, que se cierra de manera abrupta.

A lo largo del presente libro, nos hemos encontrado con errores en la ejecución de alguno de nuestros programas y hemos podido de alguna forma resolverlos, ya sea por intuición o por la búsqueda de ayuda en algún foro de la Web.

En programación existen diversos tipos de errores, sin embargo, aprenderemos a clasificarlos de acuerdo a su origen.

Java es un lenguaje orientado a objetos, esto quiere decir que todo en este lenguaje es una clase o bien un objeto, concepto que venimos remarcando desde los primeros capítulos. Entonces, cuando ocurre un error en alguna parte del código, se crea un objeto y nos conviene saber, cada vez que esto sucede, a qué clase pertenece.

Es necesario tener en cuenta las jerarquías de los distintos errores que existen. Partamos de la premisa que nos aparece un error; en principio la procedencia de este puede deberse a dos razones:

Error en tiempo de compilación	Estos son los típicos errores de sintaxis, es decir, los que cometemos cuando omitimos una llave o un punto y coma, etcétera. La solución de este tipo de errores es corrigiéndonos a nosotros mismos la forma en que escribimos el código.
Error en tiempo de ejecución	Este no depende de nosotros, pero tiene que ver con la lógica del programa; un ejemplo típico es cuando dividimos por cero. Este ocurre cuando iniciamos el play del programa y se genera un objeto.

A partir de la clase **Throwable** tenemos las jerarquías de clases que manejan los errores en Java. Esta clase a su vez se divide en dos:

Clase Error	Cuando **Error** hereda de aquí, ocurre casi siempre que es un error de hardware, en la memoria o del espacio en el disco rígido. Lo que se puede hacer es controlarlo, y avisar al usuario que nos encontramos con esta contingencia.

Clase Exception	Todos los manejos de errores se van a centrar desde aquí, puesto que estos errores sí los podemos manejar y resolver de una manera más analítica. La clase **Exception** presenta dos divisiones:
	IOException Llamadas **excepciones comprobadas**. Estas no dependen del programador, por ejemplo cuando buscamos un archivo y este no se encuentra.
	RuntimeException Llamadas **excepciones no comprobadas**. Por lo general son errores del programador, por ejemplo cuando se recorre un arreglo con más posiciones no declaradas, o errores de tipado.

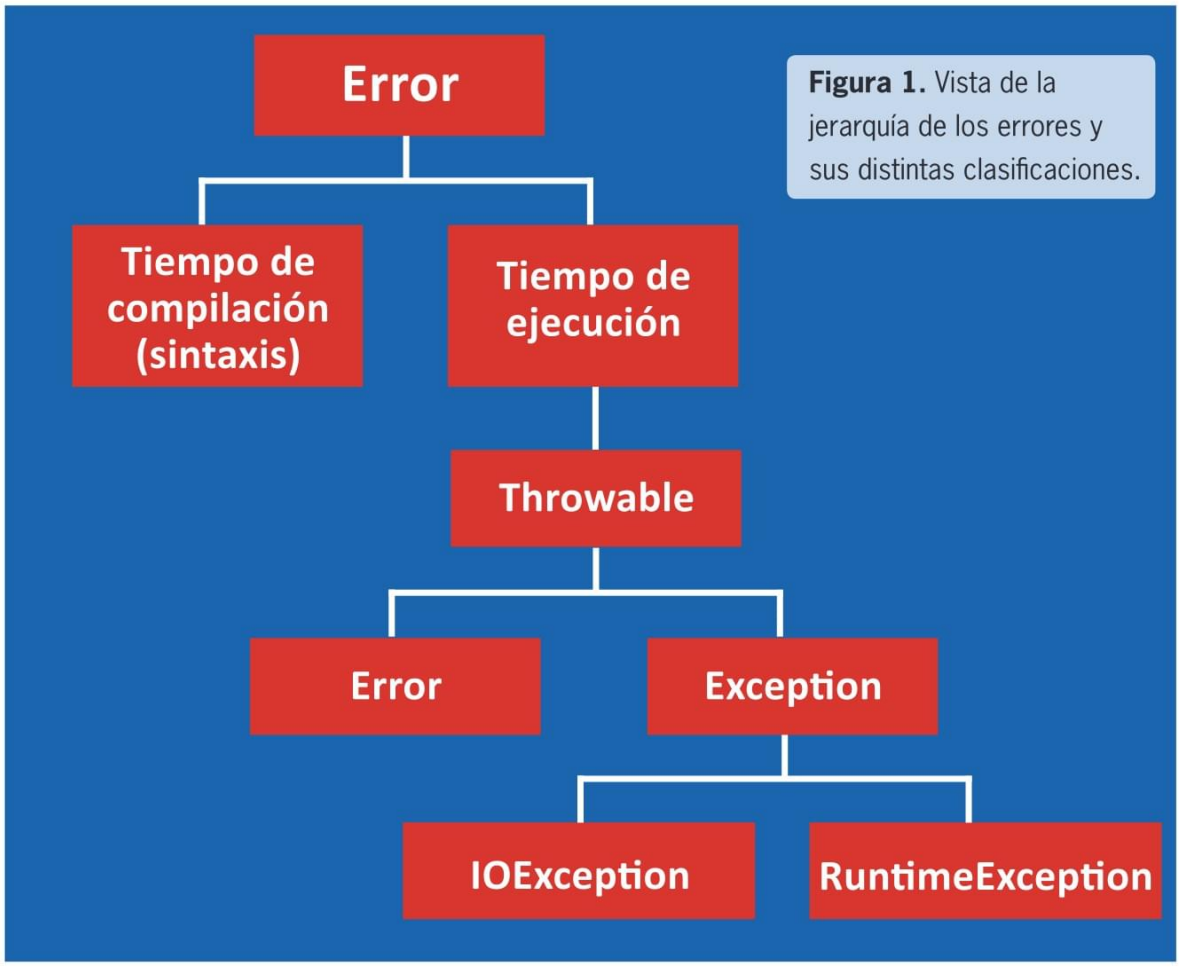


Figura 1. Vista de la jerarquía de los errores y sus distintas clasificaciones.

Veamos un pequeño ejemplo que nos servirá para explicar los distintos tipos de errores que se nos pueden presentar:

```
  package excepciones;
  import javax.swing.*;
  public class Errores {
      public static void main(String[] args) {
          int [] miMatriz = new int[5];
          miMatriz[0]= 4;
          miMatriz[1]= 14;
          miMatriz[2]= 24;
          miMatriz[3]= 34;
          miMatriz[4]= 44;

          for(int i=0; i<5;i++){
              System.out.println("Posición: " + i +" "+
miMatriz[i]);
              }

   //Petición de datos personales
     String nombre = JOptionPane.showInputDialog("Escribe tu nom-
bre");
     int edad = Integer.parseInt(JOptionPane.
showInputDialog("Ingresa tu edad"));
          System.out.println("Hola "+ nombre + " Tu edad es: "+ edad
+ " años");
          }
      }
```

Para comprobar los tipos de errores, provoquemos uno a propósito, por ejemplo, poner una llave de cierre antes de que empiece el **for**; cuando ejecutamos el programa, nos dará la siguiente salida:

```
java.lang.ClassFormatError: Duplicate field name&signature in class file excepciones/Errores
```

Ahora cometeremos un error del tipo lógico o de estructura del programa que realizamos, esta vez escribiremos una línea más a las instancias del arreglo:

```
miMatriz[5]= 54;
```

Aunque dimos espacio para **5** valores y ahora el largo son **6** valores, el IDE no marca error, pero cuando ejecutemos el programa nos dará:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5

Básicamente nos marca un error de array, que está excediendo el límite que le habíamos impuesto al inicio (**5**) y colocamos **6**. Este error es del tipo de RuntimeException, es decir, una excepción no comprobada.

De todas formas, en cualquiera de los dos casos de errores que acabamos de ejemplificar, el programa no termina de ejecutarse. Aunque el programa cuenta claramente con dos partes y, a la vez, estas nada tienen que ver una con la otra, sin embargo, no se ejecutará.

Como la que falla es la línea 11, hace que todo deje de ejecutarse, aunque nada tenga que ver el segundo bloque. Esto también puede llegar a suceder en programas con muchas líneas de código, y tal vez sea el momento en que podríamos controlar estas partes con errores y que el programa continúe ejecutándose. A esto lo denominaremos **manejo de errores y excepciones**.

Si copiamos el error en nuestro buscador de preferencia y luego elegimos la API de Java, enseguida nos daremos cuenta de que el tipo de error va a heredar de la clase **java.lang.RuntimeException**, que, a su vez, hereda de **Exception** y esta a su vez de **Throwable,** entonces queda comprobada la jerarquía.

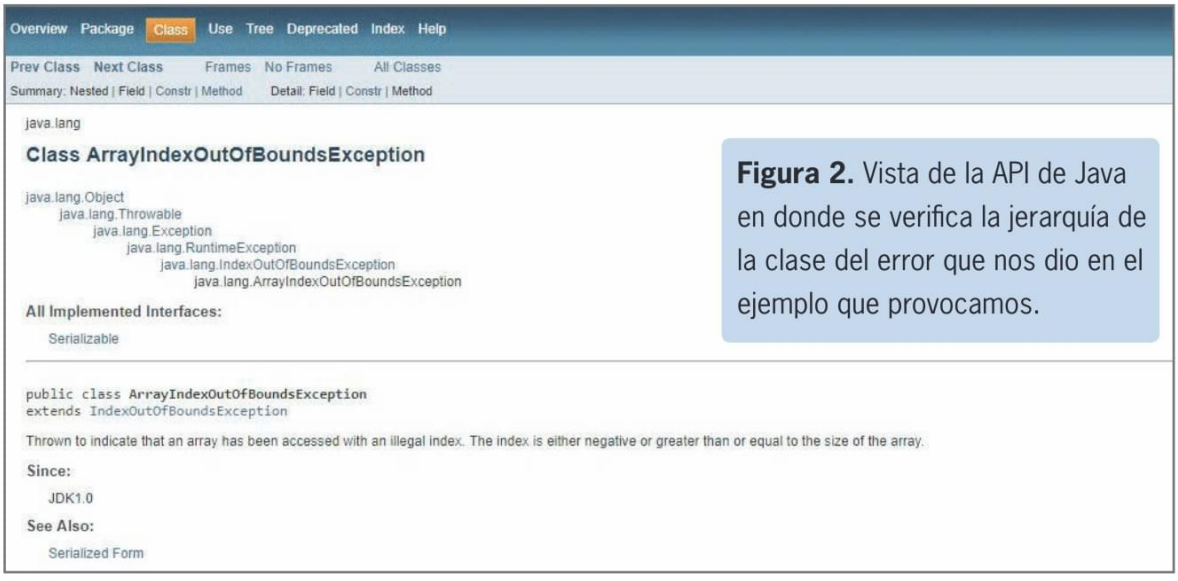


Figura 2. Vista de la API de Java en donde se verifica la jerarquía de la clase del error que nos dio en el ejemplo que provocamos.

Para el segundo tipo de errores, es decir IOException, Java, que es un lenguaje robusto, tiene ya previsto este tipo de situaciones. Esto puede suceder cuando, por ejemplo, queremos acceder a una carpeta por un archivo X; sin embargo, sin que nos enteremos, alguien pudo haber cambiado de posición o borrado el archivo, entonces esta situación provocará un error, no importa que sea ajena al programador, de allí el nombre de **excepciones no comprobadas**.

MANEJO DE EXCEPCIONES

Cuando tratemos de soslayar un error y pasarlo por alto sin que el programa se detenga, estamos hablando del **manejo de excepciones**.

Veamos un ejemplo con distintas situaciones en donde podemos observar los diferentes usos en el manejo de excepciones:

```
lecturaArchivo {
    try {
           abrir el archivo;
           verificar su tamaño;
           asignar suficiente memoria;
           leer el archivo a la memoria;
           cerrar el archivo;
    } catch (falloAbrirArchivo) {
        hacerAlgo;
    } catch (falloDeterminacionTamaño) {
        hacerAlgo;
    } catch (falloAsignaciondeMemoria) {
        hacerAlgo;
    } catch (falloLectura) {
        hacerAlgo;
    } catch (falloCerrarArchivo) {
        hacerAlgo;
    }
}
```

Supongamos entonces que, en algún lugar de nuestro programa, queremos colocar una imagen que va a ser nuestro fondo, sin embargo, esta puede que haya sido movida del lugar en donde se hallaba originalmente, entonces prepararemos un manejo para este tipo de error:

```
  class nombreClase extends ClasePadre{

      public nombreConstructor(){
             try{
               imagen = ImageIO.read(new File("src/fondos/fondo04.
png"));
//ruta en donde se debería encontrar la imagen
             }
             catch(IOException e){
             System.out.println("La imagen no se encuentra");
             }
  }
```

Por supuesto que es un error del tipo IOException, esto quiere decir que es ajeno a nosotros, sin embargo,Java contempla este tipo de situaciones y nos va a permitir que las podamos tratar. Para ello nos obligará a construir los bloques **try-catch**. Así, el programa puede seguir ejecutándose y salta el error que acaba de encontrar.

Prevención de errores

Es necesaria una lectura de la documentación de la API de Java para saber el uso correcto de los métodos en un programa. La documentación en línea especifica las excepciones que lanza un método, en el caso de que existiesen, y también nos advierte las razones por las que ellas se lanzarían. Luego, es importante documentarse de manera adecuada para conocer las clases de excepciones y las razones potenciales en las que podrían ocurrir tales eventos. Por último, hay que incluir el código adecuado para manejar esas excepciones en sus programas. Para ello, ingresamos en **https://docs.oracle.com/javase/8/docs/api/.**

EXCEPCIONES DECLARATIVAS Y NO DECLARATIVAS

Cuando hablamos de excepciones, nos encontramos con varias clasificaciones, pero existe una muy especial y es la que determina cuándo una excepción pasa a ser declarativa y cuándo no lo es, es decir, si nos tomamos el trabajo –obligados o no– de hacer un uso efectivo de las excepciones:

Excepciones no declarativas o no comprobadas	Son las que el compilador no nos obliga a tratar o atrapar. Supongamos un caso en el que se quiere implementar un usuario con un login. Sin embargo, el método login, así como está planteado, tiene un importante error de diseño: no se contempla la posibilidad de que, por algún factor externo, el método pudiera fallar, por ejemplo, que en el momento cuando se corrobora el password o el usuario, la base de datos en donde figura la información de respaldo esté caída. Entonces, como este error no está contemplado en el código, dice que la excepción no está declarada y, por lo tanto, lanzará una excepción del tipo **RuntimeException**. Así, desde un método podemos lanzarla sin tener que declararla en su origen. A su vez, cuando se llame al método, no estará obligado a encerrar la llamada dentro de un bloque **try-catch**.
Excepciones declarativas o comprobadas	Suceden cuando estamos obligados a encerrar la llamada a un determinado método dentro de un bloque **try-catch**. Si no lo hacemos, no podremos compilar. Pertenecen a la categoría del tipo **IOException**. Para esto debemos utilizar los **throws**. En esta clasificación podemos recurrir al ejemplo del archivo que no encontramos cuando este es llamado.

Cláusula throws

Throws especifica que el método puede lanzar una excepción **IOException**. Por lo que sabemos, Java requiere que los métodos capturen o especifiquen todas las excepciones declaradas que puedan ser lanzadas dentro de su ámbito.

Se puede hacer esto con la cláusula **throws** en la declaración del método.

Si tenemos en cuenta el ejemplo anterior en que no encontramos la imagen, la forma que tiene su sintaxis es la siguiente:

```
public static BufferedImage read(File input) throws IOException{
    instrucciones
}
```

Esto significa que cuando el **throws** chequea el método **read**, si todo va bien, nos devolverá un objeto del tipo **BufferedImage**; en caso contrario, es decir, si ocurre algún fallo inesperado, lanza (throw) un objeto perteneciente a la clase **IOException**, porque los errores también son considerados objetos y a este error se lo denomina **error controlado**, es decir, nos obliga a capturar un error y luego controlarlo; entonces debemos utilizar los bloques **try-catch-finally**.

BLOQUE TRY–CATCH

Esto significa que vamos a capturar un error dentro de este bloque y lo vamos a tratar. Dentro del **try**, el programa debe intentar resolver el problema y, si no lo puede hacer, capturará la excepción y realizará lo que se encuentra dentro de su bloque. La sintaxis es la siguiente:

```
try{
instrucciones}
    catch(IOException){
    instrucciones2
    instrucciones3
}
```

El bloque **try-catch** se complementa con la sección **finally**, aunque esta no sea de carácter obligatorio. Las combinaciones posibles de sus usos son **try-catch**, **try-finally** o **try-catch-finally**.

Cuando utilizamos la sección **finally**, Java nos va a asegurar que siempre, pase lo que pase, la compilación parará allí.

Veamos un ejemplo en concreto para analizar el tratamiento de las excepciones con estos dos bloques:

```
  class fondoPanel extends JPanel{

      public fondoPanel(){
             try{
               imagen = ImageIO.read(new File("src/fondos/fondo04.
png"));
             }
             catch(IOException e){
             System.out.println("La imagen no se encuentra");
             }
  }
```

En el código **try** intenta realizar la búsqueda del archivo en la ruta especificada. Si no lo encuentra, cualquiera que sea el motivo, debemos incorporar un catch, encargado de capturar la excepción y de crear una instancia de la clase **IOException**, lo llamamos **e**, y dentro del bloque colocamos lo que queremos que haga el programa, en este caso un mensaje. Con esto hemos logrado hacer una excepción chequeada.

Veamos un ejemplo en el que barajaremos varias posibilidades de que surjan errores de diversas índoles:

```
  package excepciones;
  import java.util.Scanner;
  public class Excepciones {
      public static void main(String[] args) {
          System.out.println("¿Qué deseas hacer?");
          System.out.println("1. Introducir datos");
          System.out.println("2. Salir del programa");
```

```
          Scanner entrada= new Scanner(System.in);
   int decidir = entrada.nextInt();
   if(decidir==1){
       pedirDatos();
   }else{
       System.out.println("saliendo del sistema...");
   System.exit(0);
   }
   entrada.close();
       }

   static void pedirDatos(){
       Scanner entrada = new Scanner(System.in);
       System.out.println("Ingrese su nombre");
       String nombre = entrada.nextLine();
       System.out.println("Ingrese la edad");
       int edad = entrada.nextInt();
       System.out.println("Hola "+ nombre + " tiene " + edad + "
años" );
       entrada.close();
       System.out.println("fin del programa");
   }

      }
```

Veamos las cualidades de los errores que pueden surgir a partir de comportamientos al ingresar la información requerida.

Si en vez de poner la edad ingresamos un texto, el error que nos dará es: **Exception in thread "main" java.util.InputMismatchException**.

Esto quiere decir que la excepción generada pertenece a la clase **InputMismatchException**. Ahora, lo que hay que averiguar es si este error hereda de **RuntimeException** o de **IOException**.

Cuando verificamos en la API de Java, vemos que hereda de **RuntimeException**, por lo tanto, es un error no comprobado.

Básicamente no es un error propio del programador, en este caso es del usuario, porque no ha ingresado un número. Si quisiéramos profundizar un poco más, deberíamos mejorar el código con alguna técnica de programación más avanzada.

Por ahora podemos implementarle a nuestro método que contiene el potencial error que lance una excepción del tipo que nos dio el error, luego capturarla y, si es posible, minimizar el error o, en el mejor de los casos, saltarlo. Nos quedaría de la siguiente manera:

```
  static void pedirDatos() throws InputMismatchException{

      try{
//el código que tenemos
  }catch(InputMismatchException e){
          System.out.println("Debe ingresar un valor numérico para
la edad ");
      }
 //... lo que sigue
```

Entonces incorporamos la cláusula **throws** para que verifique y cree un objeto del tipo **InputMismatchException**. Si todo va bien, debería hacer lo que se encuentra dentro del **try**; en caso contrario, captura el error, lo almacena en la instancia **e** y nos lanza una advertencia. Sin embargo, para cualquiera de los dos casos, la instrucción **System.out.println("fin del programa");** se va a ejecutar igual.

También debemos tener en cuenta que, cuando utilizamos el **throws** y le colocamos en el parámetro el tipo de error, este debe ser el mismo que le colocamos al **catch** o alguna excepción que se encuentre por encima de la jerarquía de la clase, por ejemplo:

```
  catch(Exception e){
  ...
  }
```

Entendamos que el código siempre puede ir mejorando, lo trataremos dentro del **if** inicial de tal forma que va a tener más sentido utilizar el bloque allí dentro:

```
... //el código que tenemos
  if(decidir==1){
          try{
```

```
      pedirDatos();
          }catch(InputMismatchException e){
              System.out.println("Debe ingresar un valor numérico
para la edad");
          }
      }else{
      System.out.println("saliendo del sistema...");
  System.exit(0);
  }
  entrada.close();
      }
//... lo que sigue
```

Si el usuario se decide por la opción **1**, el programa intentará ejecutar el método **pedirDatos**, si no, capturará el error y repetirá lo que hicimos en el ejemplo anterior (no olvidemos comentar o borrar el anterior **try-catch**).

Por otro lado, este tipo de excepciones, como ya hemos mencionado, heredan de **RuntimeException**, eso quiere decir que no estamos obligados a construir un bloque **try-catch**. Si sacamos el bloque, el programa se ejecutará normalmente, salvo que cuando nos pide la edad ingresemos un valor no numérico.

Generar excepciones en forma manual

Como ya hemos visto la cláusula **throws** es utilizada en la declaración de un método que podría lanzar una excepción. Ahora bien, la cláusula **throw**, de nombre muy parecido, podemos utilizarla en cualquier parte del código de tal forma que, en ese punto, se lanzará una excepción. A esto lo llamaremos una **excepción manual**.

Veamos un ejemplo:

```
/**
** @author CharlyRed
 */
import javax.swing.*;
public class CompruebaMail {
```

```
      public static void main(String[] args) {
          String mail = JOptionPane.showInputDialog("Ingresa un
mail");
          corroboraNombre(mail);
      }
      static void corroboraNombre(String mail){
          int arroba =0;
          boolean punto = false;
          for(int i=0; i<mail.length();i++){
              if(mail.charAt(i)=='@'){
                  arroba++;
              }
              if(mail.charAt(i)=='.'){
                  punto = true;
              }
          }
          if(arroba==1 && punto == true){
              System.out.println(mail + " tiene formato correcto");
          }else{
              System.out.println(mail + " NO tiene formato correc-
to");
          }
      }
  }
```

Este sencillo programa hace que nos pida una dirección de correo electrónico a través de una ventana de diálogo del **JOptionPane**, entonces, para que nos dé correcto, debemos forzosamente poner una **@** y un punto. Pero qué pasa si escribimos lo siguiente:

```
c@e.q
```

Vemos que esa casilla no tiene un formato correcto, de tal manera que debemos hacer que el programa valide aún más la entrada de los datos.

Una de las posibilidades es colocar el código dentro de un bucle para evaluar el código y, cada vez que el usuario ingrese un carácter no correspondiente, o menos de tres caracteres después de la **@** o el

punto, se pida ingresar de nuevo. La otra posibilidad es que se lance una excepción, es decir, que provoquemos un error cosa que haremos con la cláusula **throw**, sin importar en qué parte del código lo vamos a hacer. Ahora bien, el problema surge al elegir qué tipo de excepción de la API de Java se adapta al error que queremos provocar dentro de la instrucción. Para ello debemos estudiar detenidamente cada una de las excepciones y para qué sirven; sin embargo resultaría una tarea verdaderamente titánica poder analizarlas en este libro.

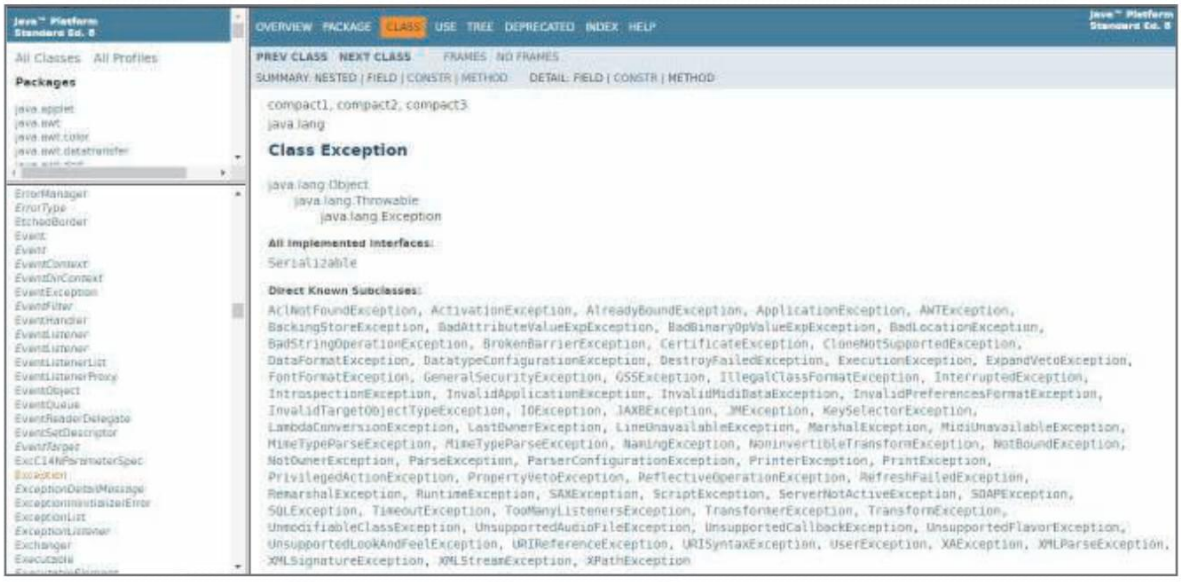


Figura 3. En la API de Java en la clase **Exception** existen una gran cantidad de excepciones y los métodos que las manejan.

Más aún, puede ocurrir que ninguna de las excepciones se adapte a lo que estamos buscando, entonces tendremos que crear nuestra propias excepciones. Para ello, utilizaremos, estos efectos, **ArrayIndexOutOfBoundsException**, veamos un ejemplo:

```
   if(mail.length()<=3){
                ArrayIndexOutOfBoundsException miExcepcion = new Ar-
rayIndexOutOfBoundsException();
                throw miExcepcion;
           }else{
   for...
//lo que está
   }
```

Utilizamos esta excepción dentro de un **if** que compara hasta **3** caracteres e instanciamos en un objeto al que llamaremos **miExcepcion**. Como **throw** captura la instancia de la clase, nos dará el error que estamos buscando, puesto que escribimos **@d.Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException**

Esto es lo que llamamos **generar una excepción manual**.

Una manera de simplificar el código anterior es escribir:

```
throw new ArrayIndexOutOfBoundsException();
```

Aquí instanciamos la clase, generando un objeto implícito.

Para que este código quede de manera completa, es decir, que no se trunque, debemos hacer la excepción en el método, para esto utilizamos la cláusula **throws**:

```
  static void corroboraNombre(String mail) throws ArrayIndexOutOf-
BoundsException{
  ...
  }

  ArrayIndexOutOfBoundsException
//al heredar esta clase de:
    java.lang.RuntimeException
     java.lang.IndexOutOfBoundsException
     java.lang.ArrayIndexOutOfBoundsException
```

Si ejecutamos el programa, no nos dará ningún error, a menos que lo provoquemos. Recordemos que, para que nosotros podamos realizar este tipo de excepciones no controladas (que heredan de **RuntimeException**), al hacer este tipo de operaciones no estamos obligados a construir una excepción con el bloque **try-catch**.

Sin embargo, dejar esto como está no se considera una buena práctica, para ello debemos adaptar nuestra excepción a un código mejor.

Nos correremos a la otra clasificación de los errores, hablamos de las **IOException**, que sí son controladas, y sí podemos capturar la excepción.

Escribimos en la línea del **throw** la siguiente clase de excepción, que tiene que ver con el tratamiento de archivos o de elementos externos, por ejemplo, lo que escribimos en el cuadro de diálogo de entrada del **mail**:

```
throw new EOFException();
```

El IDE empieza a dar sugerencias, puesto que esta clase hereda de las excepciones controladas, es decir, en primer lugar debemos importar la biblioteca de este tipo, hacemos lo que nos pide, **Alt + Enter** para hacer la importación.

Luego el error va a ser en la línea:

```
corroboraNombre(mail);
```

Si nos detenemos a analizar, verificamos que nos indica que es una excepción controlada, es decir, forzosamente debemos construir el bloque **try-catch**.

```
...
try{
        corroboraNombre(mail);
    }catch (EOFException e) {
            System.out.println("El mail no es correcto");
    }
...
```

La diferencia que existe al utilizar la cláusula **throw** es que esa excepción ocurre de manera forzosa, puesto que lo hacemos manualmente, y esto sucede cuando se llega a un final inesperado y podemos manipularlo a nuestra mejor conveniencia.

Errores adaptativos

Aunque el compilador no implementa el requerimiento de atrapar o declarar para las excepciones no verificadas, nosotros debemos proporcionar un código apropiado para este tipo de circunstancias que sabemos que pueden ocurrir. Por ejemplo, un programa debería ser capaz de procesar la excepción **NumberFormatException** del método **ParseInt** de **Integer**, aun cuando esta excepción sea del tipo excepciones no verificadas, puesto que es una subclase de **RuntimeException**. Esto hace que nuestros programas sean más robustos.

EXCEPCIONES PROPIAS

Cuando las excepciones que utilizamos en la API de Java no se adaptan a lo que realmente estamos tratando de incorporar a nuestro código para el manejo de errores, nos vemos obligados a crear nuestras propias excepciones.

Como las excepciones son clases, crearemos una clase, pero esta heredará de alguna de las clases más bien generales: **Exception**, **IOException** (con ellas estamos obligados a construir un bloque **try-catch**) o **RuntimeException** (aquí no estamos obligados a capturar la excepción).

A partir del ejemplo anterior, reharemos nuestro código y crearemos una clase que herede de la clase **Exception**:

```
class miExcepcion extends Exception{
    }
```

Esta tendrá dos constructores:

```
...
public miExcepcion(){

    }

public miExcepcion(String msjError){
        super(msjError);
    }...
```

Cuando creamos este tipo de clases, debemos crear dos constructores: uno sin parámetros (por si no queremos mostrar ningún mensaje) y el otro que contenga algún parámetro.

Una vez creada la clase **miExcepcion**, ya podemos ir al método en donde es posible lanzar la excepción:

```
    static void corroboraNombre(String mail) throws miExcepcion{
...
}
```

Luego debemos utilizar la cláusula **throw** dentro del **if**:

```
  …
  if(mail.length()<=3){
            throw new miExcepcion("El mail debe contener por lo me-
nos 3 caracteres");

          } …
```

Hemos colocado dentro de los parámetros un mensaje personalizado.
Nuevamente la línea:

```
  corroboraNombre(mail);
```

Contiene el error que necesita forzosamente el bloque **try-catch**.
Notemos algo que ya a esta altura del capítulo deberíamos saber: si en vez de colocar **miExcepcion** colocamos **RuntimeException**, el error anterior desaparece, puesto que como sabemos no precisa del bloque mencionado. En primer lugar, importamos lo que se encuentra dentro de **io.***:

```
  import java.io.*;
```

Ahora escribiremos el bloque **try-catch**:

```
  public static void main(String[] args) {
            String mail = JOptionPane.showInputDialog("Ingresa un
mail");

          try{
          corroboraNombre(mail);
      }catch (Exception e) {
              System.out.println("El mail no es correcto");
      }
      }
      static void corroboraNombre(String mail) throws miExcepcion{
  …
  }
```

Una de las maneras que usan los programadores es dejar un mensaje referido a porqué han utilizado una excepción; para ello consignan la siguiente instrucción después de la salida:

```
...
System.out.println("El mail no es correcto");

            e.printStackTrace();
      ...
```

Cuando ejecutamos nos dará el siguiente mensaje:

```
El mail no es correcto excepciones.miExcepcion: El mail debe contener por lo menos 3 caracteres.
```

Por un lado nos muestra el mensaje que colocamos, es decir, que ha entrado en el **catch** y, por otro, nos advierte que ha ocurrido una excepción del tipo que indica.

CAPTURA DE VARIAS EXCEPCIONES

Una práctica muy común consiste en manejar varias excepciones a la vez siempre que, después del bloque **try**, tengamos varios bloques **catch**.

Para poder incluir este tipo de bloques múltiples, recrearemos un ejemplo, típico en las excepciones, que se da cuando en el denominador colocamos cero y nos da la excepción del tipo aritmética: **java.lang.ArithmeticException: / by zero**.

Pasará a darnos otra excepción cuando, en vez de introducir un valor numérico, pongamos un texto o un carácter:

```
lang.NumberFormatException: For input string:"s".
```

```
package excepciones;
/**
```

```
   * @author CharlyRed
   */
  import javax.swing.*;
  public class VariasExcepciones {
      public static void main(String[] args) {
          division();
      }
      static void division(){
          int num1 = Integer.parseInt(JOptionPane.
showInputDialog("Ingrese un número"));
          int num2 = Integer.parseInt(JOptionPane.
showInputDialog("Ingrese el divisor"));
          System.out.println("El resultado es: " + num1/num2);

      }
  }
```

Como podemos observar, sea cual fuere el error, existen varios tipos de excepciones, por lo tanto, en primer lugar debemos ver qué tipo de excepción es, y esto lo hacemos desde la API de Java. Luego verificamos que ambas heredan de una clase **RuntimeException**, es decir, son excepciones no controladas. Si hacemos el tradicional bloque de **try-catch**:

```
  ...
  try{
          division();
      }catch (Exception e){
              System.out.println("Ocurrió un error");
      }
  ...
```

Cuando ejecutamos, resuelve (a medias) la excepción; pero no podremos saber de qué se trata, pues como ya hemos dicho, hay varias excepciones en este ejemplo y no habría forma de especificarlas.

Entonces, la mejor forma de trabajar este tipo de inconvenientes es capturarlos uno a uno; lo haremos de la siguiente forma:

```
        try{
           division();
       }catch (ArithmeticException e){
               System.out.println("no debes dividir por cero");
       }catch(NumberFormatException e){
               System.out.println("Debes ingresar un número");
       }
```

Como vemos en el código anterior, se ha introducido un bloque **catch** para la captura de la excepción aritmética y, a continuación, se ha puesto otro bloque **catch** para la excepción del formato numérico, cada uno con su respectivo mensaje de error.

Además de mostrar al usuario un mensaje de error personalizado, Java nos provee de tres tipos de mensajes (métodos) de sus propias bibliotecas (de la clase **Throwable** y otro de la clase **Object**), veamos:

```
  (NumberFormatException e){
      System.out.println("Debes ingresar un número");
      System.out.println(e.getMessage());
```

El método **getMessage** –de la clase **Throwable**- nos indicará que el error es del tipo **String**:

```
Debes ingresar un número
For input string:"s"
```

También podemos colocar los siguientes métodos que son de la clase **Object** y que resultan aún más específicos:

```
   System.out.println("Se ha generado un error del tipo: " +
e.getClass().getName());
```

Cuando ejecutamos y provocamos el error, nos dará:

```
Debes ingresar un número
Se ha generado un error del tipo: java.lang.NumberFormatException
```

BLOQUE FINALLY

El bloque **finally** no es obligatorio y, si está presente, debe colocarse luego del último bloque **catch**; si estos no existieran, debería ir luego del bloque **try**.

Hay que agregar como característica que el bloque **finally** se ejecutará, se lance o no una excepción.

Vemos un ejemplo sencillo que nos pedirá que elijamos una opción para que halle el área:

```
  package excepciones;
  /* @author CharlyRed
   */
  import javax.swing.*;
  import java.util.*;
  public class ClausulaFinally {
      public static void main(String[] args) {
      Scanner teclado = new Scanner(System.in);
          System.out.println("\n----Hallaremos el área de las
siguientes figuras---------");
      System.out.println("Elija una opción: \n1: Cuadrado \n2:
Triángulo \n3: Rectángulo");
      figura = teclado.nextInt();
      teclado.close();

      switch (figura){
          case 1:
              int lado = Integer.parseInt(JOptionPane.
showInputDialog("Ingrese el lado"));
              System.out.println(Math.pow(lado,2));
              break;
          case 2:
              int base = Integer.parseInt(JOptionPane.
showInputDialog("Ingrese la base"));
              int altura = Integer.parseInt(JOptionPane.
showInputDialog("Ingrese la altura"));
              System.out.println("El área del triángulo es: " +
(base * altura)/2);
```

```
                break;
            case 3:
                base = Integer.parseInt(JOptionPane.
showInputDialog("Ingrese la base"));
                altura = Integer.parseInt(JOptionPane.
showInputDialog("Ingrese la altura"));
                System.out.println("El área del rectángulo es: " +
(base * altura));
                break;
            default:
                System.out.println("la opción no es la correcta");
        }

        }
        static int figura;

    }
```

En este caso será necesario utilizar este bloque **finally**, ya que el programa, aun cuando es muy pequeño, consume recursos innecesariamente. Además provocaremos errores, por ejemplo, del tipo de formato cuando en el panel de diálogo ingresemos un carácter; entonces nos veremos obligados a capturar las excepciones que se van sucediendo en el programa.

```
...
try{
    figura = teclado.nextInt();
    teclado.close();
    }catch (Exception e){
        System.out.println("Ha surgido un error");
    }
...
```

Con esto logramos salvar el primer error, y así el programa seguirá funcionando. De todas maneras, aunque el programa siga para adelante o se trunque mostrando el mensaje de error, el scanner no se cierra y seguirá consumiendo recursos. El código correcto es el siguiente:

```
...
try{
     figura = teclado.nextInt();
     }catch (Exception e){
          System.out.println("Ocurrió un error");
     }finally{
          teclado.close();
     }
...
```

Con esto, **finally** hará que el teclado se cierre sin importar si el programa entra a una excepción o si todo está correcto.

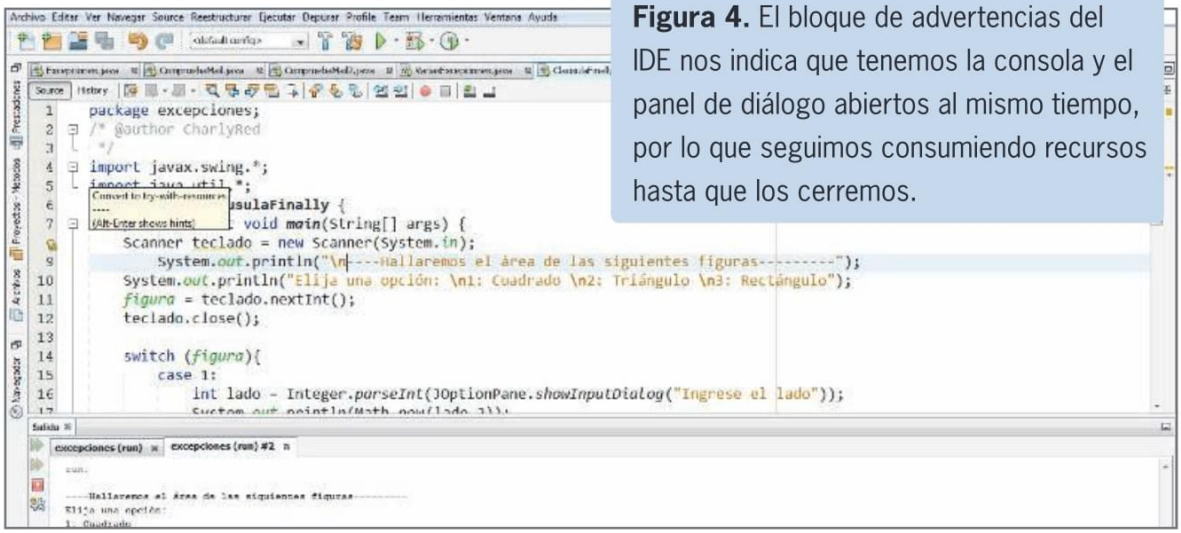


Figura 4. El bloque de advertencias del IDE nos indica que tenemos la consola y el panel de diálogo abiertos al mismo tiempo, por lo que seguimos consumiendo recursos hasta que los cerremos.

RESUMEN CAPÍTULO 02

En este capítulo vimos las excepciones, muy útiles a la hora de capturar los errores y tratarlos debidamente para que nuestros programas sigan funcionando. Conocimos la cláusula throw, vimos la diferencia entre ellas y la forma de usarlas, también analizamos el uso correcto de los bloques try-catch, esenciales a la hora de hacer las capturas y el tratamiento de los errores de distinta índole. Creamos, asimismo, nuestras propias excepciones. Para terminar vimos la cláusula finally, elemento imprescindible cuando hay que finalizar eventos y que nuestros programas no consuman recursos innecesarios.

Actividades 02

Test de Autoevaluación

1. ¿Qué son las excepciones? ¿Qué tipos existen?

2. ¿Cómo hacemos un manejo correcto de ellas? ¿Qué recaudos debemos tener en cuenta para su buen uso?

3. En cuanto a la jerarquía de las excepciones, ¿por qué es tan importante conocerlas? ¿Qué papel juega la API de Java para ello?

4. ¿Qué diferencia hay entre la clase RuntimeException y la IOException?

5. ¿Qué son las excepciones declarativas y las no declarativas? ¿Qué diferencia existe entre ellas?

6. ¿Cuál es el uso efectivo de la cláusula throws? ¿Tiene alguna diferencia con la cláusula throw? Explique el uso de esta última.

7. ¿Cómo se usa el bloque try-catch? ¿Cómo encierra los códigos que queremos tratar?

8. ¿Cómo creamos una excepción personalizada?

9. Si un programa es susceptible de tener varios tipos de errores, ¿cómo podemos generar la captura de estos?

10. ¿Cuándo usamos el bloque finally? ¿Es obligatorio hacerlo?

Ejercicios prácticos

1. Cree una calculadora simple y capture los errores, por ejemplo, las divisiones por cero.

2. Recree un pequeño programa, de tal forma que pida el ingreso de datos.

Luego el programa debe ser capaz de evitar errores de ingreso de datos.

Recursividad

La recursividad es uno de los métodos más avanzados en lo que respecta a las estructuras de datos. En este capítulo presentaremos ejemplos para resolver los clásicos factoriales y las torres de Hanói. Otro de los ejemplos que trataremos es la serie de Fibonacci. Además, abordaremos otros temas, como las pilas y las llamadas a métodos; también compararemos la recursividad con la iteración, y explicaremos las ventajas y desventajas entre un método y otro.

03

CONCEPTOS INICIALES

En las ciencias de la computación, el uso de la **recursividad** es uno de los temas más reiterados, puesto que abundan casos, como el de la teoría de los números y la resolución de problemas, bastante complejos y algunos irresolutos para esta rama de la computación –que es la computación cuántica–. Uno de ellos es el tratamiento de los **qbits**. No es que vayamos a resolver ese formidable asunto, ni mucho menos, pero si trataremos de entender algunas cuestiones profundas y enigmáticas de la ciencia de la computación, y más aún de la estructura de datos.

Básicamente la recursividad se da cuando un método se llama a sí mismo un número n de veces y debemos, a través de un **algoritmo**, controlar este número de veces; por lo tanto un problema puede ser resuelto en función de su tamaño.

Como un método se llama a sí mismo, en consecuencia, se va a asignar un espacio en la pila para las nuevas variables y los parámetros. Luego de volver, se tiende a recuperar de esa pila las variables locales y los parámetros anteriores, por ende, la ejecución se va a reanudar en el punto en que iniciamos la llamada al método.

Para solucionar estos métodos recursivos, debemos tener en cuenta que poseen una gran cantidad de elementos en común y que, resolviendo de a pequeñas instancias, lograremos solucionar un gran problema. A estas pequeñas instancias, las denominaremos **caso base**, que tendrá la respuesta explícita al problema; luego, se encuentra el **dominio**, que divide el problema en problemas más pequeños (**problema-1**) hasta llegar al caso base.

La recursividad resulta un concepto difícil de entender en un principio, pero, luego de analizar diferentes problemas, veremos que aparecen siempre puntos en común. Veamos los primeros.

Programación recursiva

Cuando se decide abordar una subrutina recursiva, se necesita definir un caso base, luego se definirán las reglas para subdividir los casos más complejos en el caso base. Para una subrutina recursiva es esencial que, con cada llamada recursiva, el problema se reduzca de forma que al final se llegue al caso base.

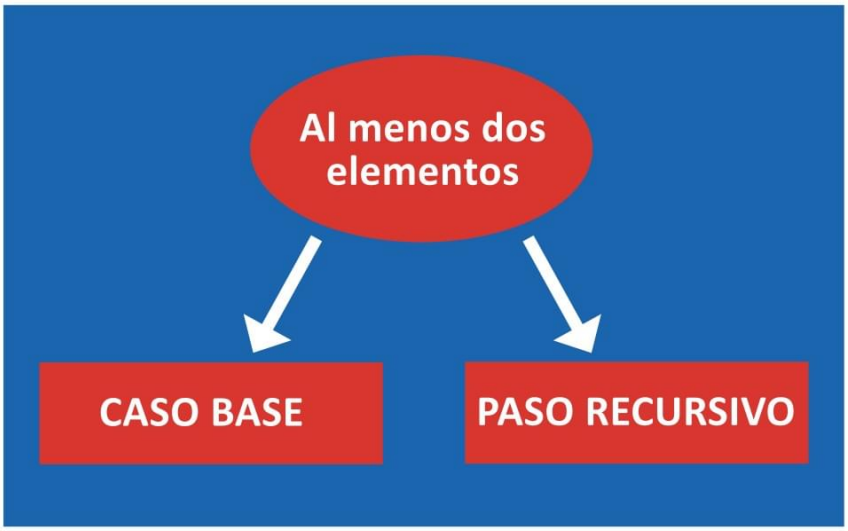


Figura 1. Vista de la validez de un método recursivo. Por lo menos deben existir dos elementos: el caso base y el paso recursivo propiamente dicho, llamado también **dominio**.

Veamos un sencillo ejemplo, que nos servirá para explicar los distintos tipos de errores que se nos pueden presentar:

```
package recursividad;

public class Recursividad {
void repetir(){
    repetir();
}
    public static void main(String[] args) {
        Recursividad recur = new Recursividad();
        recur.repetir();
        }
 }
```

El método **repetir** es recursivo, porque se llama a sí mismo (lo dice el concepto). Cuando ejecutamos el programa, este se bloqueará y va a generar la siguiente excepción:

```
Exception in thread "main" java.lang.StackOverflowError
at recursividad.Recursividad.repetir(Recursividad.java:7)...
```

Se repite tantas veces hasta que la pila se llene y se cuelgue.

Como podemos ver, el programa ejecuta el método **main**, y hay un objeto instanciado **recur**, el que llama al método **repetir**.

Debemos tener en cuenta que, cada vez que hacemos el llamado a un método, se reservarán 4 bytes, y estos se liberarán cuando finalice la ejecución.

Luego, esta operación se va a repetir hasta que la **pila estática** se llene y colapse en detrimento del sistema, que se "colgará".

Ahora analicemos este programa:

```
public class Recursividad2 {
    void imprimir(int num){
        System.out.println(num);
        imprimir(num-1);
    }
    public static void main(String[] args) {
        Recursividad2 recur2 = new Recursividad2();
        recur2.imprimir(5);
    }
}
```

El **main** llama al método **imprimir**, al que le envía el valor **5**. El parámetro **num** recibe este valor y, luego, lo imprime. Como le pedimos que **num-1**, entonces llamará nuevamente al método e imprimirá; de esta forma seguirá haciéndolo por lo menos 9700 veces (según la capacidad de la memoria que tengamos en nuestra PC), y el programa se bloqueará.

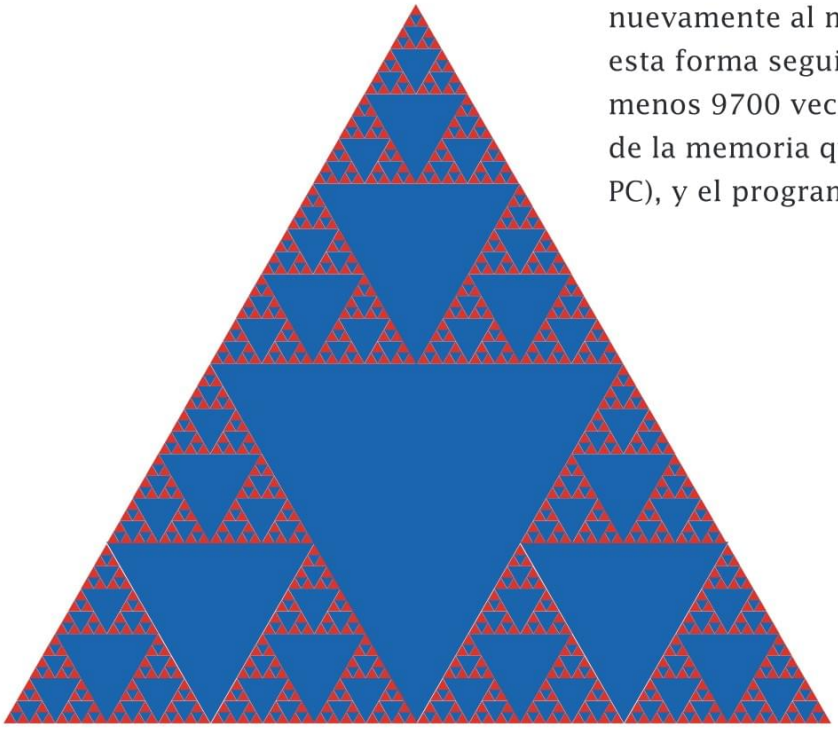


Figura 2. Figura de un fractal de triángulos equiláteros. De sus lados se van a construir triángulos más pequeños hasta que se tornen infinitos y casi invisibles.

TIPOS DE RECURSIVIDAD

Existen varios tipos de recursividad:

Recursión directa o simple

Ocurre cuando el código **X** tiene una sentencia que involucra a **X**. Con esto aparecerá una sola llamada recursiva. Es muy sencillo trasladar estos códigos a iteraciones.

Recursión indirecta o cruzada

Ocurre cuando la función **X** involucra una función **Y** que invoca a la vez una función **Z**, y así sucesivamente, hasta que se involucra la función **X**. Por ejemplo, el algoritmo de par o impar, o las torres de Hanói.

```
package recursividad;
import java.util.Scanner;
public class ParImpar {
    public static void main(String[] args) {
        int numero;
        Scanner entrada = new Scanner(System.in);
        System.out.print("Introduce el número: ");
        numero = entrada.nextInt();
        if(par(numero))
        System.out.println("el número es  par");
        else
        System.out.println("El número es impar");
        }

     public static boolean impar (int numero){
     if (numero==0)
      return false;
     else
      return par(numero-1);
 }

 public static boolean par (int numero){
     if (numero==0)
      return true;
```

```
        else
         return impar(numero-1);
    }
}
```

Si un número **X** es par o impar, entonces devolverá un mensaje; en caso contrario, otro.

Recursión múltiple

Se da cuando hay más de una llamada a sí misma dentro del cuerpo de la función, resultando más difícil de transformar a iterativa. Por ejemplo, el algoritmo de la serie de Fibonacci.

Recursión anidada

En algunos de los argumentos de la llamada hay una nueva llamada a sí misma. Como ejemplo tenemos la función de Ackermann.

Recursión infinita

La iteración y la recursión pueden producirse infinitamente. Un bucle infinito ocurre si la prueba o test de continuación del bucle nunca se vuelve falsa. Dicha función se ejecutará hasta que la computadora agote la memoria disponible y se produzca una terminación anormal del programa.

```
public  void cicloInfinito(Long x){

    System.out.println(" " + x);
    cicloInfinito(x + 1);
}

public static void main(String[] args) {

    StackOverFlow stackOvF = new StackOverFlow();
    stackOvF. cicloInfinito (0L);
}
```

Veremos a continuación algunos de los clásicos ejemplos en los que podemos usar métodos recursivos.

EJEMPLOS DE RECURSIVIDAD

Existen muchos ejemplos en los que se emplea la recursividad. Uno bastante particular es el siguiente: supongamos que nos encontramos con una casa muy grande, al mejor estilo de las mansiones de las películas de terror. Entonces, ingresamos a una de las habitaciones y nos damos cuenta de que dentro de ella hay varias puertas; al entrar por una de ellas, nos encontramos con una nueva cantidad de puertas, y así sucesivamente hasta que nos vemos realmente perdidos. Pero, de alguna manera, uno de los pasillos nos lleva al punto inicial, es decir, a la primera puerta. Para solucionar este problema laberíntico, es posible realizar un mapeo de la casa, que solo necesitaremos hasta que nos encontremos en el punto inicial.

Factoriales

Ahora veamos un ejemplo más real, el caso de los factoriales. Sabemos que el **factorial** de un número es n!. Donde n es un número que se va a multiplicar sucesivamente hasta llegar a 1, de la siguiente manera:

```
4! = 4x3x2x1 = 24      -> donde 24 es el 4!
```

Veamos cómo encontramos el factorial de **4** sin usar recursividad:

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

       int fact = 1;
                 for(int i=1;i<=4;i++){
                 fact *= i;

     }
    System.out.println(fact);
}
}
```

Utilizamos un bucle **for** que va a recorrer hasta el **4** y luego imprime **fact**.

Veamos cómo se realiza el mismo ejercicio, pero esta vez utilizando la recursividad:

```
public class FactorialR {
    public static int factorial(int n){
    if (n == 0){
            return 1;
    }else{
            return n * factorial(n-1);
          }
  }

 public static void main(String[] args) {
   FactorialR far = new FactorialR();
   int f = far.factorial(4);
   System.out.println("El factorial de 4 es: " + f);
}
}
```

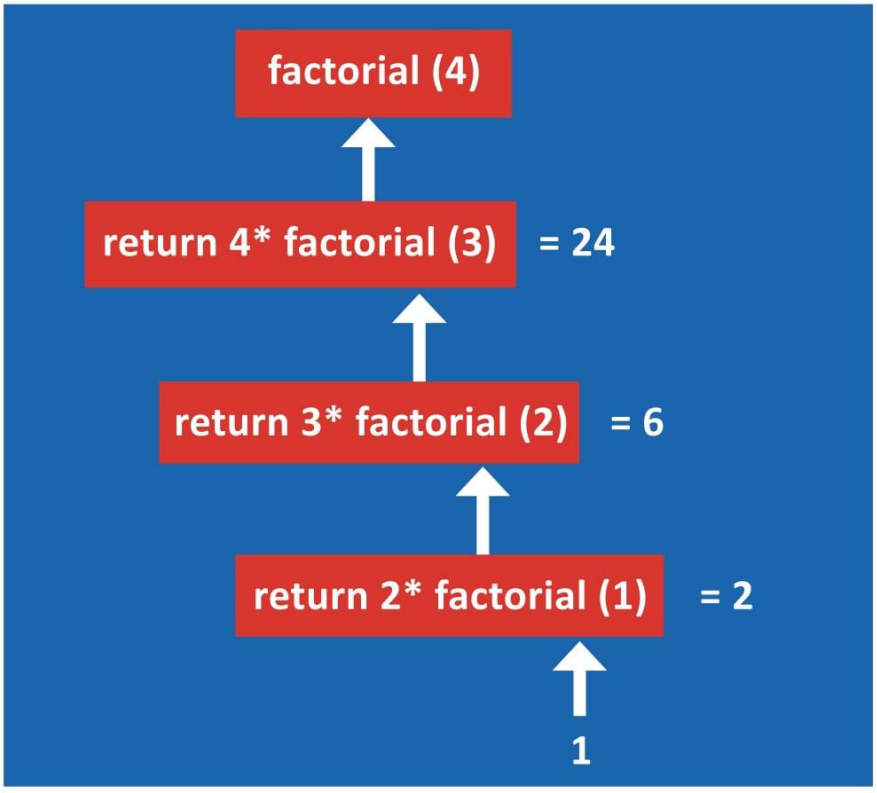


Figura 3. Dado que para hallar un factorial se sigue un patrón, es decir, el de multiplicar al número por el factorial de ese número -1, se puede optar por la recursividad.

El método **factorial** es recursivo porque, desde este mismo método, estamos llamando al mismo método **factorial**, es decir a sí mismo.

Hagamos el seguimiento del problema.

Sabemos que el factorial de un número **n** sigue esta secuencia **n × (n-1) × (n-2) ×...× 2 × 1**.

Dado que este problema sigue un patrón repetitivo, tiende a ser más fácil de resolver mediante el uso de los métodos recursivos.

También podríamos decir que el factorial de un número **n** es equivalente al número **n** multiplicado por el factorial **n-1**.

La sucesión de Fibonacci

Esta curiosa sucesión tiene la siguiente forma: 0, 1, 1, 2, 3, 5, 8, 13... Como podemos observar, empieza de 0 y 1, y luego se va sumando la última cifra con la anterior para dar el nuevo número; y esto sucede hasta el infinito. Una curiosidad que suscita esta secuencia es que la proporción entre esos dos sumandos va a dar un valor constante de 1,618... A este número se lo conoce como **proporción dorada o áurea**.

Para definir esta secuencia de manera recursiva, debemos aplicar la siguiente sintaxis:

```
fibonacci (0) = 0;
fibonacci (1) = 1;
fibonacci (n) = fibonacci (n-1) + Fibonacci (n-2);
```

Notemos que aquí existen particularmente dos casos base, cuando **fibonacci (0) = 0** y cuando **fibonacci (1) = 1**.

Veamos entonces el código recursivo para resolver esta secuencia cuando ingresemos la cantidad que necesitemos que tenga la sucesión:

```
package fibonacci;
import java.util.Scanner;
public class FibonacciRecursivo {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
```

```
            System.out.print("Ingrese el número de elementos a mostrar
de la serie: ");
            int limite = sc.nextInt();
            sc.close();

         for(int i = 0; i<limite; i++){
         System.out.print(funcionFibonacci(i) + ", ");
            }
       }

       private static int funcionFibonacci(int num){
            if(num == 0 || num==1)
                 return num;
            else
          return funcionFibonacci(num-1) + funcionFibonacci(num-2);
       }
      }
```

Casos base (de salida, aquí no se produce llamada recursiva):

f(0) = 0
f(1) = 1

Caso recursivo:

f(n) = f(n-1) + f(n-2)

Como era de esperarse, la **funcionFibonacci** es llamada a sí misma, de tal forma que resuelve las sumatorias sucesivas del último número más el anterior. Si quisiéramos resolver esta sucesión a través de una solución no recursiva, es decir, por iteración, el código debería ser:

```
import java.util.Scanner;
public class FibonacciNR {
     public static void main(String[] args) {

         int n1 = 0;
         int n2 = 1;
```

```
            Scanner sc = new Scanner(System.in);
            System.out.print("Ingrese el número de elementos a mostrar
de la serie: ");
            int limite = sc.nextInt();

            System.out.print(n1 + ", ");
            System.out.print(n2 + ", ");

            for(int i = 0; i<limite-2; i++){
                n2 = n1 + n2;
                n1 = n2 - n1;
                System.out.print(n2 + ", ");
            }
        }
    }
```

Aquí, como en el caso de los factoriales, hemos implementado el uso de un iterador, como es el **for**, dentro del que colocamos el condicional.

Una curiosidad que hay que resaltar es que entre ambos casos, pongamos como ejemplo un caso extremo, se puso el número **50**. Si iteramos, el programa ejecuta en 2 segundos, mientras que si lo hacemos por el método de recursividad, tarda 3'10". En ocasiones, una llamada recursiva se genera por llamadas repetidas y no se considera una solución efectiva, puesto que hay que repetir el cálculo de los valores inferiores para las distintas ramas del resultado. En el siguiente tema abordaremos las ventajas de utilizar uno u otro método.

Función de Ackermann

El algoritmo de **Ackermann** es muy famoso en las ciencias de la computación, este nos dice: si se toman dos números naturales como argumentos iniciales, nos devuelve un único número natural. Veamos la lógica de este algoritmo:

$$A(m,n) = A(m-1, A(m,n-1)) \qquad m>0 \wedge n>0$$
$$A(0,n) = n+1 \qquad m=0$$
$$A(m,0) = A(m-1,1) \qquad m>0 \wedge n=0 \quad (0,1)$$

```
Akm(m, n)
    if(m==0)
    return n+1
    else
    if(n==0 ^ m>0)
    return Akm(m-1, 1)
    else(
    return Akm(m-1, A((m, n-1))
    end if
    end if
    end if
fin Akm


import java.util.Scanner;
public class Ackermann {
    static Scanner entrada = new Scanner(System.in);

     static int Ackermann(int m,int n){
        if(m==0)
        return n+1;
        else if(n==0)
        return Ackermann(m-1,1);
        else
        return Ackermann(m-1, Ackermann(m,n-1));
}

    public static void main(String[] args) {
        int m,n;
   System.out.println("Ingrese un número natural:");
        m = entrada.nextInt();
  System.out.println("Ingrese otro número natural:");
        n = entrada.nextInt();
  System.out.print(Ackermann(m,n) + " es el valor del natural re-
sultante.");
      }
}
```

El programa hace uso de una función recursiva de Ackermann encargada de hallar el valor Ackermann entre dos números. Se lee el valor de los dos números y luego se invoca la función recursiva.

NÚMEROS DE A(m,n)

M/N	0	1	2
0	1	2	3
1	2	3	4
2	3	5	7
3	5	13	29
4	13	65533	$2^{65533} - 3 \approx 2.10^{19728}$
5	65533	A (4,65533)	A (4,A (5,1))
6	A (5,1)	A (5,A (5,1))	A (5,A (6,1))

M/N	3	4	*n*
0	4	5	n + 1
1	5	6	n + 2
2	9	11	2n + 3
3	61	125	8.2n - 3
4	A (3,2^{65536} - 3)	A (3,A (4,3))	$2^{2^{...2}}$ - 3 (n + 3 términos)
5	A (4,A (5,2))	A (4,A (5,3))	
6	A (5,A (6,2))	A (5,A (6,3))	

Figura 4. Números de A(m,n). En general, por debajo de la fila 4, ya no es posible escribir todos los dígitos del resultado de la función.

Recursividad y la pila de llamada a métodos

Analicemos cómo funciona un programa en el proceso de ejecución de la pila recursiva. Bajo una llamada recursiva el sistema reserva espacio, denominado **tabla de activación**, donde se debe almacenar una copia de los objetos locales y los parámetros de la subrutina en ese momento.

La tabla de activación se apila sobre las llamadas recursivas anteriores, formando lo que se conoce como **pila recursiva**. Este proceso termina cuando un nuevo valor del parámetro no produce una nueva llamada recursiva, esto quiere decir que se alcanza al caso base.

Una vez alcanzada esta situación, el sistema va liberando el espacio reservado conforme las subrutinas se van ejecutando sobre su tabla de activación. Finaliza con la **llamada inicial**.

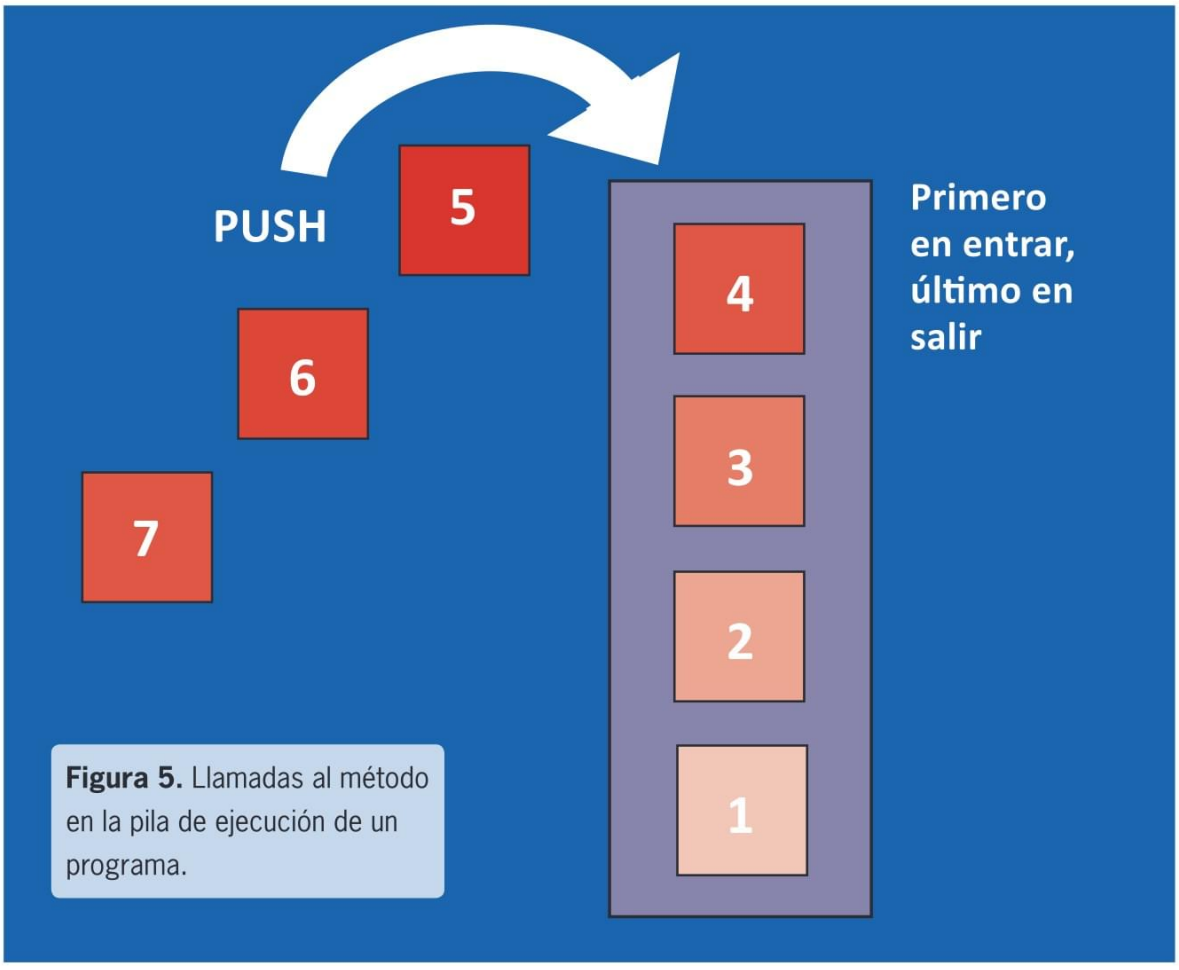


Figura 5. Llamadas al método en la pila de ejecución de un programa.

RECURSIVIDAD VERSUS ITERACIÓN

Hemos analizado antes algunos ejemplos clásicos en los que se pueden utilizar ambos métodos; sin embargo, ahora veremos los pros y los contras que surgen al programar de la forma iterativa o bien recursiva.

Ambos métodos se basan fundamentalmente en una instrucción de control: la iteración utiliza una instrucción de repetición (do-while, while o for), en tanto que la recursividad utiliza instrucciones de selección (if, if-else o switch).

Tanto la iteración como la recursividad van a implicar la repetición: la primera la utiliza de forma explícita y la segunda hace repeticiones a través de llamadas sucesivas al método.

Ambos implican una prueba de terminación, lo que significa que la iteración termina cuando falla la condición de continuación del ciclo, mientras que en la recursividad termina cuando se llega al caso base.

Mientras que la iteración y la recursividad llegan a la terminación de manera gradual y controlada, en la primera se producen modificaciones en el contador hasta que asuma el valor de la condición para terminar con el ciclo, y en la segunda se van produciendo versiones cada vez más pequeñas del problema original.

Ambos métodos pueden volverse infinitos. Sin embargo, en la iteración, si se vuelve falsa pierde la continuidad, en tanto que en la recursión infinita esto ocurre si no se reduce el problema hasta llegar a converger con el caso base o si el caso base no se evalúa.

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

  Hanoi  objHanoi = new Hanoi();
  objHanoi.torresHanoi(3,1, 2, 3);
  System.out.println("Juego completado");

      }
// creando el método recursivo
      public void torresHanoi(int discos, int torre1, int torre2, int
torre3){
            // caso base
            if(discos ==1){
                  System.out.println("Mover Disco de Torre " +
torre1 + " a Torre " + torre3 );
            }else{
                  //Dominio
                  torresHanoi(discos-1, torre1, torre3, torre2);
                  System.out.println("Mover Disco de Torre " +
torre1 + " a Torre " + torre3);
                  torresHanoi(discos-1, torre2, torre1, torre3);

            }
      }
  }
```

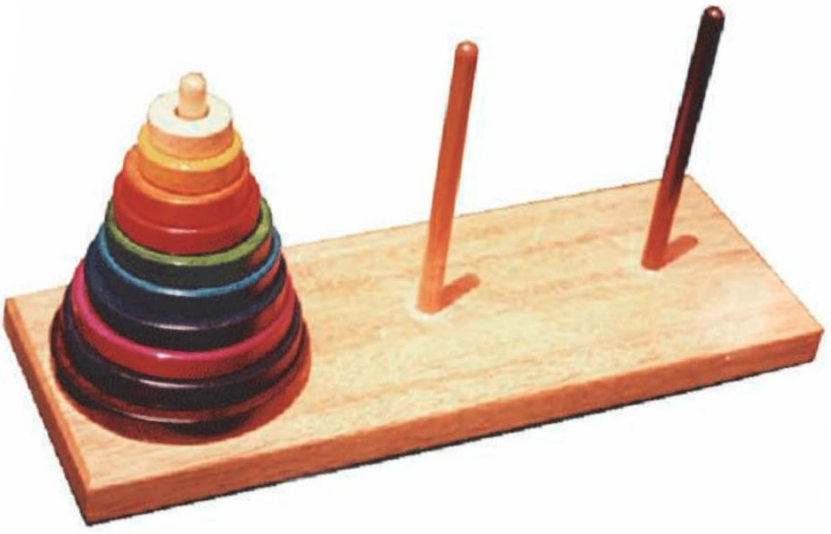


Figura 6. En la figura se muestra este juego matemático, muy utilizado en psicología cognitiva en niños.

Verificamos que el método que hace la recursión **torresHanoi**, previa comparación en un **IF**, se va autoinvocando y a la vez va moviendo los discos, cumpliendo las reglas del juego. Finalmente al llegar al caso base, el juego ha sido resuelto.

Para ver cómo funciona este algoritmo, podemos visitar la dirección **www.disfrutalasmatematicas.com/juegos/torre-de-hanoi-2.html**.

RESUMEN CAPÍTULO 03

Hablar de recursividad es tocar temas que tienen que ver con la teoría de los números, quizás uno de los más apasionantes de las matemáticas y tal vez uno de los más difíciles. En este capítulo hemos tocado conceptos de lo que significa la recursividad y el uso dentro de las soluciones a algoritmos comunes. Analizamos varios tipos de ellos y abordamos también ejemplos clásicos, como la función de Ackermann o la sucesión de Fibonacci. También vimos las pilas y las llamadas sucesivas a los métodos, y diferenciamos lo que es utilizar la recursividad o hacerlo de la manera tradicional iterativa. Finalmente, recreamos un algoritmo para solucionar un curioso juego matemático, hablamos de las torres de Hanói.

Actividades 03

Test de Autoevaluación

1. ¿Qué es la recursividad?

2. ¿Cómo hacemos un manejo correcto de este tipo de métodos?

3. ¿Qué clases de recursividad existen? Cite algunos de los ejemplos más relevantes que conozca. Visite webs y analice varios de ellos.

4. Releve algunas de las diferencias entre hacer recursividad o iteración en ejemplos como factoriales.

5. Haga lo mismo con la función de Ackermann o Fibonacci.

6. ¿Qué es la pila de llamada a métodos? ¿Por qué es necesario abordar este tipo de recurso?

7. ¿Qué se entiende por el algoritmo recursivo en ejemplos como la torres de Hanói? ¿Podría recrear el ejemplo sin usar recursividad?

8. ¿Qué ventajas y desventajas ocurren cuando utilizamos métodos recursivos en lugar de las clásicas iteraciones?

9. ¿De qué manera llamamos a un método recursivo y cuándo debemos dar por sentado que hemos llegado al final o caso base?

10. ¿A qué llamamos explosión de llamada a métodos?

Ejercicios prácticos

1. Recree una sucesión con sumas sucesivas. Luego modifique el código, pero esta vez con restas sucesivas.

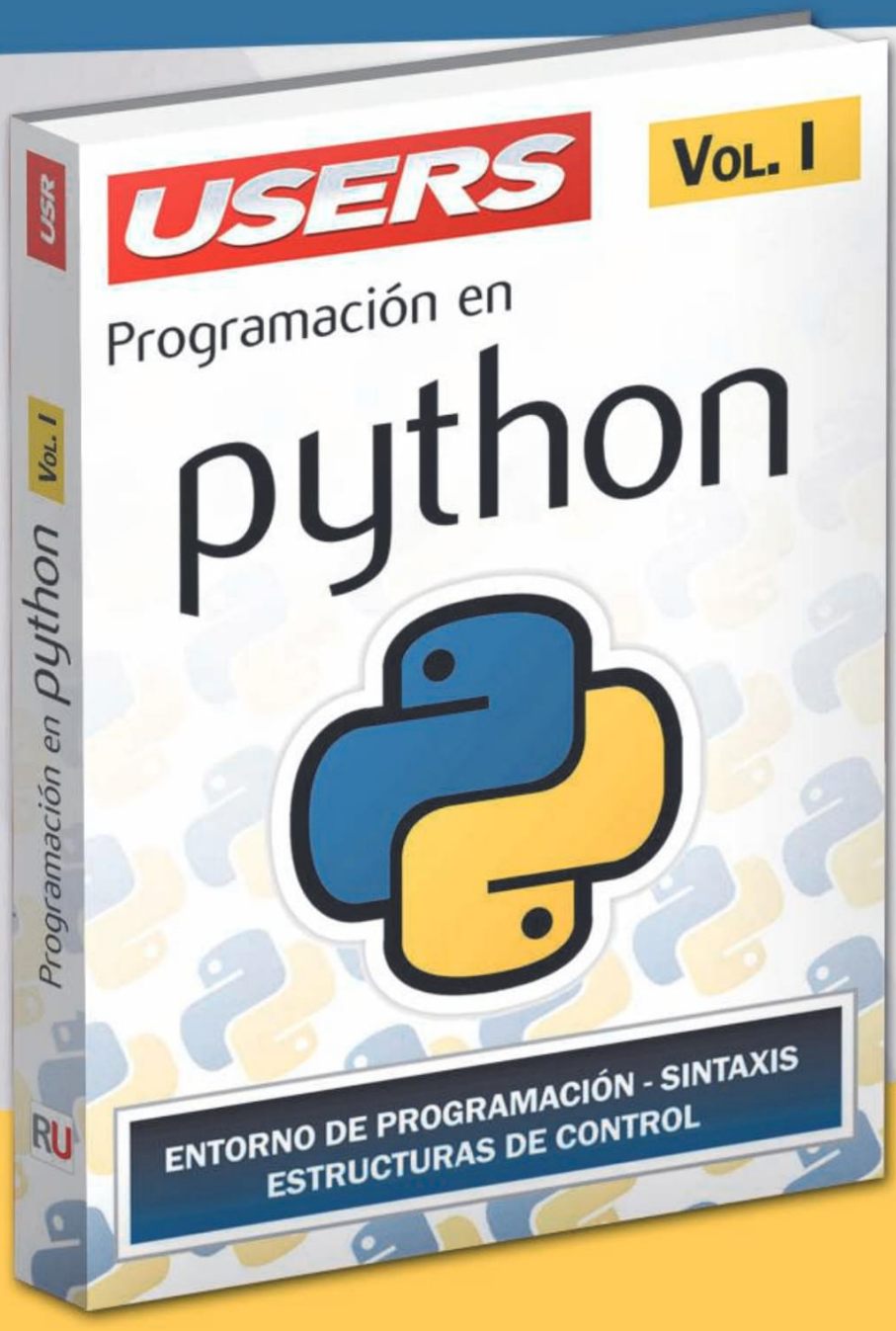
2. Programe un algoritmo recursivo para resolver el cuadrado latino.

3. Realice un algoritmo recursivo y otro no recursivo para mostrar el mayor y el menor número de un vector.

4. Codifique con recursión la sumatoria de los elementos de una matriz cuadrada de 5x5.

5. Realice un método recursivo para hacer una tabla de multiplicar. También realice el ejercicio utilizando iteraciones.

CURSO DE PROGRAMACION PYTHON



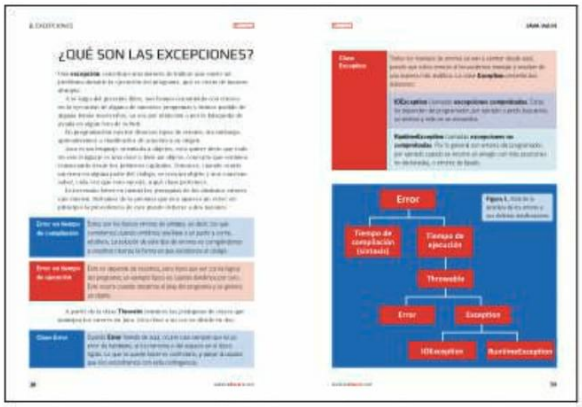
USERS

Programación en Java™

Vol. III

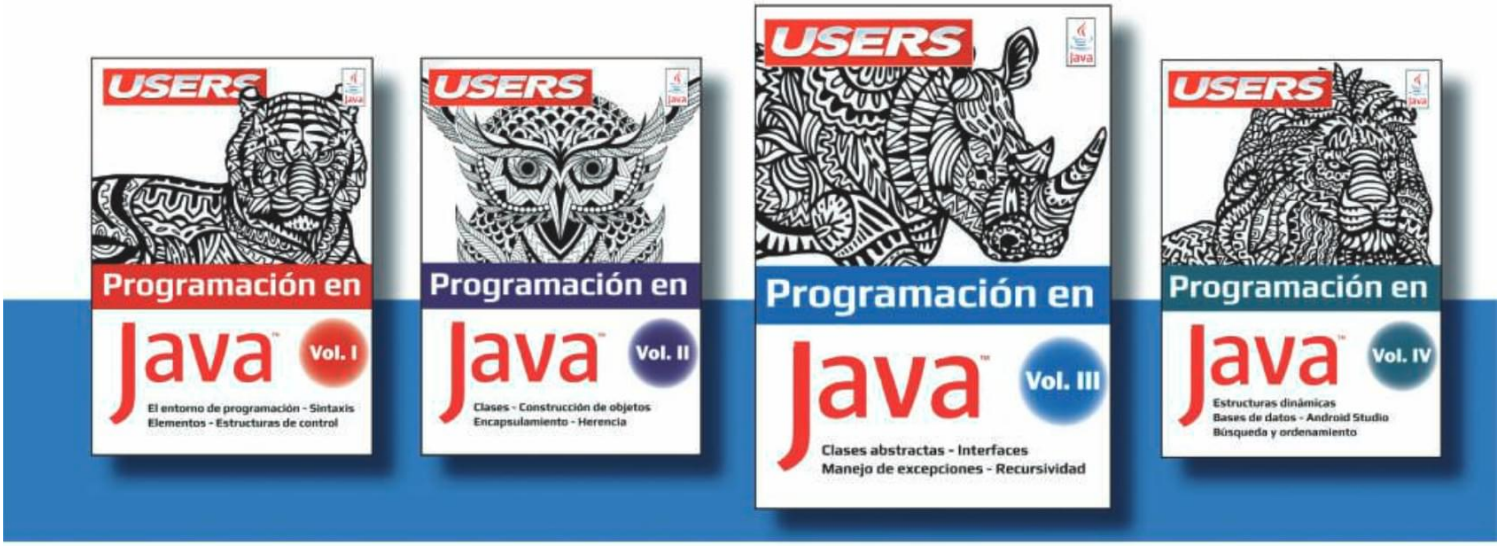
ACERCA DE ESTE CURSO

Java es uno de los lenguajes más robustos y populares en la actualidad, existe hace más de 20 años y ha sabido dar los giros adecuados para mantenerse vigente. Este curso de Programación en Java nos enseña, desde cero, todo lo que necesitamos para aprender a programar y, mediante ejemplos prácticos, actividades y guías paso a paso, nos presenta desde las nociones básicas de la sintaxis y codificación en Java hasta conceptos avanzados como el acceso a bases de datos y la programación para móviles.



ACERCA DE ESTE VOLUMEN

En este volumen se presentan las clases abstractas e interfaces, excepciones y recursividad. Aprenderemos a crear y utilizar interfaces, revisaremos el manejo y captura de excepciones y analizaremos algunos ejemplos de recursividad.



SOBRE EL AUTOR

Carlos Arroyo Díaz es programador, escritor especializado en tecnologías y docente. Se desempeña como profesor de Informática General, Java y Desarrollo Web. También ha trabajado como mentor docente en el área de Programación en varios proyectos del Ministerio de Educación de la Ciudad Autónoma de Buenos Aires.

RedUSERS

En nuestro sitio podrá encontrar noticias relacionadas y participar de la comunidad de tecnología más importante de América Latina.

RedUSERS PREMIUM

RedUSERS PREMIUM la biblioteca digital de USERS. Accederás a cientos de publicaciones: Informes; eBooks; Guías; Revistas; Cursos. Todo el contenido está disponible online - offline y para cualquier dispositivo. Publicamos, al menos, una novedad cada 7 días

ISBN 978-987-4958-11-2