

Los constructores sobrecargados

Para crear objetos en Java se utilizan los constructores. Java permite que existan más de un constructor para cada clase. Cada constructor se diferencia de los demás mediante la lista de parámetros. Debido a que Java define un constructor por defecto sin parámetros para cada clase, cada vez que agreguemos un constructor necesariamente este deberá tener diferentes parámetros. A estos constructores se los denomina constructores sobrecargados.

La sobrecarga es un término para indicar que la semántica de un constructor o un método es la misma (es decir el nombre del constructor o el nombre del método), diferenciándose en la firma de ese constructor o método (es decir la lista de parámetros).

Normalmente los constructores sobrecargados se utilizan para inicializar las variables miembro de un objeto.

El siguiente ejemplo ayudará a identificar todos los conceptos vertidos hasta ahora


Ejemplo: Defina una clase denominada DVDPelícula y cree una clase de tal manera que al momento de su creación asigne los siguientes valores a ese objeto:

Item	Valor
Nombre del objeto	copiaDVD
Nombre de película	El Robo del Siglo
Fecha de estreno	5 de marzo de 2020
Recaudación	6514000 dólares
Género	Cine policiaco
Sinopsis	Viernes 13/01/06. Los policías aguardan la voz de su jefe. Miguel Sileo, el negociador, espera hablar con Vitette, uno de los líderes de la banda de ladrones que entró a la sucursal del Banco Río.
Codigo	123456
Stock	25
Disponible para alquiler	Verdadero

```

1 package ar.edu.unju.fi.modelo;
2 import java.text.SimpleDateFormat;
3 import java.util.Date;
4
5 public class DVDPelícula {
6     private int codigo;
7     private String nombre;
8     private Date fechaEstreno;
9     private double recaudacion;
10    private String genero;
11    private String sinopsis;
12    private byte stock;
13    private boolean disponibleAlquiler;
14
15    public DVDPelícula() {
16    }
17    public DVDPelícula(int codigo, String nombre, Date fechaEstreno, double recaudacion,
18        String genero, String sinopsis, byte stock, boolean disponibleAlquiler) {
19        this.codigo = codigo;
20        this.nombre = nombre;
21        this.fechaEstreno = fechaEstreno;
22        this.recaudacion = recaudacion;
23        this.genero = genero;
24        this.sinopsis = sinopsis;
25        this.stock = stock;
26        this.disponibleAlquiler = disponibleAlquiler;
27    }
28
29    public String mostrarDatos() {
30        SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yy");
31        String fechaString = dateFormat.format(fechaEstreno);
32        return String.format("Codigo: %d\tNombre: %s\tFecha Estreno: %s", codigo,nombre,fechaString);
33    }
34 }

```





En la definición anterior se puede observar en el recuadro **A** se han definido dos constructores para la clase DVDPelícula.

La diferencia entre ellos radica en la lista de parámetros (a eso se denomina firma) ya que el nombre de los constructores siempre es igual al nombre de la clase.

Observe además que en esta ocasión se ha utilizado el constructor sobrecargado para enviar por parámetro el valor de cada una de las variables miembro.

La siguiente porción de código permite completar el funcionamiento del ejemplo:

```
1 package demoJSE;
2
3 import java.util.Calendar;
4
5 import ar.edu.unju.fi.modelo.DVDPelicula;
6
7 public class Principal {
8
9     public static void main(String[] args) {
10         Calendar calendario = Calendar.getInstance();
11         calendario.set(2020, 2, 5);
12         DVDPelicula copiaDVD = new DVDPelicula(123456, "El Robo del Siglo",
13                                             calendario.getTime(), 6514000d,
14                                             "Cine Policiaco", "Viernes ...",
15                                             (byte)25,true);
16
17         System.out.println(copiaDVD.mostrarDatos());
18
19     }
20
21 }
```

Se puede observar que se envía por parámetro todos los valores de las variables miembro del objeto.

Aquí se puede notar varias cosas:

- 1) Se recuerda que en versiones previas al JDK 8 para asignar un valor a un objeto de tipo `Date`, se debe usar `Calendar` o `SimpleDateFormat` (en este caso se utilizó `Calendar`). `Calendar` es una clase abstracta (tema que se encara más adelante), por tanto, para crear el objeto no se puede utilizar el operador `new`. La propia clase `Calendar` provee el método `getInstance()` que crea el objeto. Para asignar el valor concreto de una fecha al calendario se utiliza el método `set(año, mes, día)`, donde los meses varían del 0 al 11.
- 2) Por otro lado, al enviar el valor del `stock` se lo escribe de la siguiente manera `(byte) 25`. Esto se debe a que el tipo de dato de la variable miembro `stock` es `byte` y ya se ha comentado que por defecto los valores literales Java los interpreta como `int`. En la siguiente sección se estudiará la conversión explícita.

Finalmente, en la línea 17 se invoca al método `mostrarDatos()` del objeto `copiaDVD` para obtener un `String` que se muestra por pantalla, y cuyo resultado es:

```
Codigo: 123456 Nombre: El Robo del Siglo Fecha Estreno: 05/03/20
```

Los métodos sobrecargados

Como se mencionó en el apartado anterior, la sobrecarga hace alusión a porciones de código que poseen la misma semántica, pero diferente firma. En el caso de los métodos hace referencia a **un objeto** (no clase como es el caso de los constructores) que posee dos o más métodos con el mismo nombre (semántica) pero diferente lista de parámetros (firma).

Veamos como se visualizan estos métodos sobrecargados en un ejemplo integrador.

Ejemplo: Se desea poder calcular el área de un Círculo ya sea por su radio, o pasándole el radio de cualquier otro círculo.

Veamos que significan estos requerimientos dentro de la definición de la clase en el lenguaje Java

```
1 package ar.edu.unju.fi.modelo;
2
3 public class Circulo {
4     private double radio;
5
6     public Circulo() { //constructor por defecto
7
8     }
9
10    public Circulo(double radio) { //constructor sobrecargado
11        this.radio = radio;
12    }
13
14    public double calcularArea() {
15        return Math.PI*Math.pow(this.radio, 2);
16    }
17
18    public double calcularArea(double radio) {
19        return Math.PI*Math.pow(radio, 2);
20    }
21
22    public double getRadio() {
23        return radio;
24    }
25
26    public void setRadio(double radio) {
27        this.radio = radio;
28    }
29 }
```

Se observan:

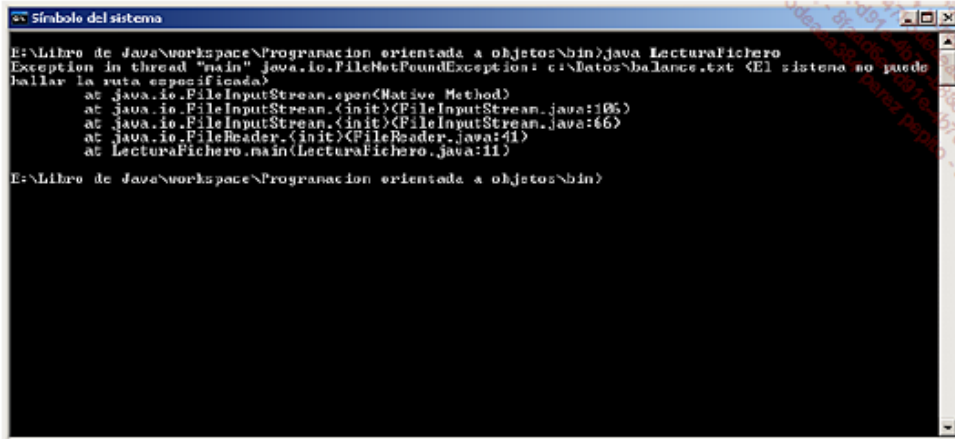
- 1) Se define la clase Circulo
- 2) Posee una variable miembro privada (línea 4) para definir el radio del círculo
- 3) Posee constructores sobrecargados (remarcados con línea roja). Observe que estos métodos poseen el mismo nombre (semántica) pero difieren en la lista de parámetros (firma). Observe además que el método definido entre líneas 14 y 16 obtiene el área del objeto usando la variable miembro radio, se puede afirmar que este método “Devuelve el área de ese círculo”; mientras que el método definido entre líneas 18 y 20 devuelve el área usando como valor el parámetro, por esto se puede afirmar que “Devuelve el área de algún círculo cuyo radio fue pasado por parámetro”. Si bien el objetivo de ambos métodos es el mismo, ese objetivo difiere del objeto de cálculo.

La gestión de excepciones

Los errores en Java se pueden clasificar en tres categorías:

1. Los errores de sintaxis. Este tipo de errores se produce en tiempo de compilación cuando una palabra clave del lenguaje está mal escrita. Son habituales en ambientes de desarrollo donde el editor de código y el compilador son dos entidades separadas ya que generalmente en los IDE tales como Eclipse, NetBeans, Jbuilder, entre otros proporcionan un análisis sintáctico al mismo tiempo que se escribe el código, provocando que este tipo de errores se minimice al momento de compilar las clases. Por otro lado, los IDE no solo detectan este tipo de errores, sino que sugieren solucionar para corregirlos.

- Los errores de ejecución. Estos errores aparecen después de la compilación, más precisamente cuando se realiza la ejecución de la aplicación. La sintaxis del código es correcta, pero el entorno de la aplicación determina que no permite ejecutar alguna instrucción empleada en la misma. Ejemplos típicos de este tipo de errores son: intenta abrir un archivo que no existe en el disco de su máquina. Sin duda, esto generará un problema y la aplicación no podrá continuar ejecutándose. En el mejor de los casos cuando Java detecte este tipo de errores generará un mensaje. Para el ejemplo planteado se presentaría el siguiente mensaje



Además, Java permite que el desarrollador programe la recuperación ante este tipo de errores. Para ello utiliza las excepciones, que serán el objeto de estudio en este apartado.

- Errores de lógica. Es el error menos deseado, ya que el programa compila sin problema y se ejecuta sin generar errores, pero no funciona como estaba previsto. En este caso, hay que revisar la lógica de funcionamiento de la aplicación. Las herramientas de depuración nos permiten seguir el desarrollo de la aplicación, situar puntos de interrupción, visualizar el contenido de las variables, etc. Estas herramientas no sustituyen sin embargo una buena dosis de reflexión.

Las Excepciones

Cuando se produce un error durante la ejecución de un método, se crea un objeto `Exception` que permite representar el error que acaba de producirse. Este objeto contiene numerosa información relativa al error ocurrido en la aplicación, así como el estado de esta en el momento en que se presentó el error. A continuación, se transmite y notifica la presencia de este objeto a la JVM. Esto activa la excepción y entonces, la JVM intenta buscar una solución para resolverla. Para ello, explora los diferentes métodos mediante las invocaciones realizadas hasta alcanzar la ubicación donde se produjo el error. Además, por cada invocación la JVM registra si el método invocado posee un gestor de excepciones capaz de tratar el problema. La búsqueda empieza con el método en el cual se activó el error y, a continuación, sube hasta el método que ha disparado la ejecución de la aplicación, si es necesario. Cuando se localiza un gestor de excepciones adecuado, se le transmite el objeto `Exception` para que se encargue de su tratamiento. Si la búsqueda no da resultado, la aplicación se detiene.

Clasificación de excepciones

Las excepciones se suelen clasificar en tres categorías.

- Las excepciones verificadas corresponden a una situación anormal durante el funcionamiento de la aplicación. Esta situación suele estar relacionada con algún

elemento exterior a la aplicación, como por ejemplo una conexión hacia una base de datos o la lectura de un archivo.

Estas excepciones se representan mediante instancias de la clase `Exception` o una de sus subclases. Deben tratarse, obligatoriamente, dentro de un bloque **`try catch`** o propagarse al código que la invoca mediante la palabra clave **`throws`** declarada en la firma del método invocante.

2. Los errores corresponden a condiciones excepcionales exteriores a la aplicación que esta última no puede prever. Estas excepciones se representan mediante una instancia de la clase `Error` o una de sus subclases. Estas excepciones no tienen por qué tratarse. Cabe precisar que cuando se producen el funcionamiento de la aplicación se ve, por lo general, comprometido. Si se gestionan, la mayor parte del tiempo el único procesamiento consiste en mostrar un mensaje algo menos alarmante al usuario del que le propone por defecto la máquina virtual de Java. En prácticamente todos los casos es inevitable que se detenga la aplicación.
3. Los errores relacionados con un uso incorrecto de alguna funcionalidad del lenguaje, o un error de lógica en el diseño de la aplicación. El error más frecuente que podrá encontrar cuando se inicia en el desarrollo con Java será sin duda la excepción `NullPointerException`, que se produce cuando se intenta utilizar una variable del tipo Referencia no inicializada. Estas excepciones se representan mediante una instancia de la clase `RuntimeException`. Si bien estas excepciones podrían procesarse dentro de bloques **`try catch`**, no se recomienda hacerlo. Es preferible analizar el código y modificarlo para evitar que aparezcan.

Recuperación de excepciones

La gestión de las excepciones ofrece la posibilidad de proteger un bloque de código contra las excepciones que podrían producirse en él. Las porciones potencialmente susceptibles de generar un error deben ubicarse dentro un bloque **`try`**. Si se produce una excepción dentro de este bloque de código, el o los bloques de código **`catch`** son examinados. Si alguno es capaz de tratar la excepción, se ejecuta el código que contiene, en caso contrario, la misma excepción se activa para eventualmente ser recuperada por un bloque **`try`** de mayor nivel. Una instrucción **`finally`** permite marcar un grupo de instrucciones que se ejecutarán, ya sea a la salida del bloque **`try`** si no se produjo ninguna excepción, o a la salida de un bloque **`catch`** si se produjo alguna una excepción. Por lo tanto, la sintaxis general es la siguiente:

```
try
{
...
Instrucciones peligrosas
...
}
catch (excepción1 e1)
{
...
código ejecutado si se produce una excepción de tipo Excepción1
...
}
catch (excepción2 e2)
{
...
código ejecutado si se produce una excepción de tipo Excepción2
...
}
finally
{
...
código ejecutado en cualquier caso antes de la salida del bloque try
o de un bloque catch
...
}
```



Para cada bloque `catch` es necesario definir el tipo de excepción que se gestionará. Analicemos esta afirmación en el siguiente ejemplo

```
public void leerFichero(String nombre)
{
    FileInputStream fichero=null;
    BufferedReader br=null;
    String línea=null;
    try
    {
        fichero=new FileInputStream("c:\\Datos\\balance.txt");
    }
    catch (FileNotFoundException e)
    {
        e.printStackTrace();
    }
    br=new BufferedReader(new InputStreamReader(fichero));
    try
    {
        línea=br.readLine();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
    while (línea!=null)
    {
        System.out.println(línea);
        try
        {
            línea=br.readLine();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

El primer **try** envuelve una porción de código que intenta abrir un archivo en la ubicación "c:/datos/balance.txt" para almacenar su contenido dentro del objeto denominado `fichero`. Esta situación puede llegar a generar una excepción debido a que podría suceder que el archivo no exista en la ubicación indicada. Para este bloque **try** se define un bloque **catch** que captura la excepción correspondiente: `FileNotFoundException`, que representa la excepción que se dispara cuando se presenta el error de que el archivo indicado no existe. Observe, además, que en el bloque **catch** se define el objeto `e` que es de tipo `FileNotFoundException`. Este objeto posee un método denominado `printStackTrace()` que permite obtener todo el historial desde el momento en que se invoca el método que disparó la excepción, brindando de esta manera una forma de seguir la forma en que se presentó el error.

Como se puede observar en este ejemplo, cada instrucción susceptible de producir una excepción puede protegerse mediante su propio bloque **try**. Esta solución presenta la ventaja de ser extremadamente precisa para la gestión de las excepciones en detrimento de la legibilidad del código.

Una solución más sencilla y elegante consiste en agrupar varias instrucciones en un mismo bloque **try** y definir una sucesión de bloques **catch**. Esta forma de representar las excepciones permite mantener el código es más legible, pero a cambio se pierde precisión de la instrucción que provoca la excepción.

También hay que tener cuidado con el orden de los bloques **catch** ya que deben ser organizados siempre desde el más preciso hasta el más general. Las excepciones, al ser clases, pueden tener relaciones de herencia (tema que se estudiará más adelante). Si se prevé un bloque **catch** para



gestionar un tipo particular de excepción, éste puede también gestionar todos los tipos de excepciones que heredan de ella. Por este motivo es que los bloques **catch** deben estar organizados para que las excepciones más específicas se puedan ejecutar primero.

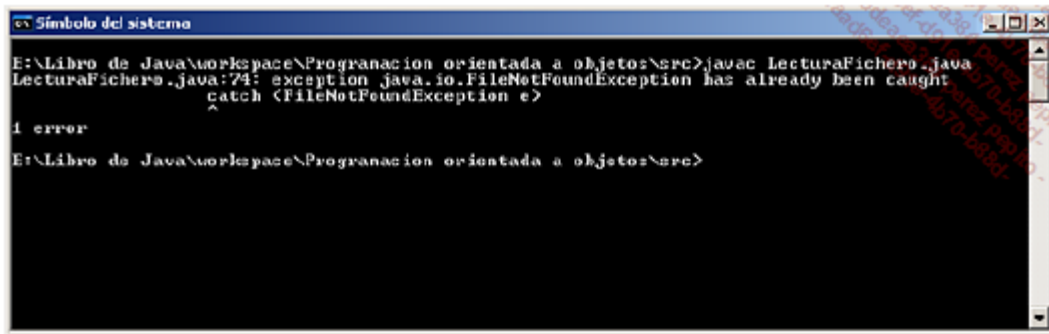
El siguiente código organiza el ejemplo anterior usando un bloque **try** y varios bloques **catch**

```
public void leerFichero(String nombre)
{
    FileInputStream fichero=null;
    BufferedReader br=null;
    String línea=null;
    try
    {
        fichero=new FileInputStream(nombre);
        br=new BufferedReader(new InputStreamReader(fichero));
        línea=br.readLine();
        while (línea!=null)
        {
            System.out.println(línea);
            línea=br.readLine();
        }
    }
    catch (FileNotFoundException e)
    {
        e.printStackTrace();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}
```

En este ejemplo nuestro ejemplo ya que la clase `FileNotFoundException` hereda de la clase `IOException`. Si bien no se ha definido que significa que una clase herede de otra, esta afirmación significa que el compilador detectará que el orden de los bloques `catch` es el correcto. Si por el contrario se definiera el código de la siguiente manera:

```
public void leerFichero(String nombre)
{
    FileInputStream fichero=null;
    BufferedReader br=null;
    String línea=null;
    try
    {
        fichero=new FileInputStream(nombre);
        br=new BufferedReader(new InputStreamReader(fichero));
        línea=br.readLine();
        while (línea!=null)
        {
            System.out.println(línea);
            línea=br.readLine();
        }
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
    catch (FileNotFoundException e)
    {
        e.printStackTrace();
    }
}
```

el compilador generará el siguiente error:



El código puede ser todavía más preciso si se indica que un mismo bloque **catch** debe gestionar varios tipos de excepciones. Los distintos tipos de excepciones que puede procesar un bloque **catch** deben indicarse en la declaración separados mediante el carácter |.

Observe el siguiente ejemplo:

```
public void leerFichero(String nombre)
{
    FileInputStream fichero=null;
    BufferedReader br=null;
    String línea=null;
    double suma=0;
    try
    {
        fichero=new FileInputStream(nombre);
        br=new BufferedReader(new
InputStreamReader(fichero));
        línea=br.readLine();
        while (línea!=null)
        {
            System.out.println(línea);
            línea=br.readLine();
            suma=suma+Double.parseDouble(línea);
        }
        System.out.println("total:"+suma);
    }
    catch (IOException | NumberFormatException e)
    {
        e.printStackTrace();
    }
}
```

Aquí se puede observar que el bloque catch gestiona dos excepciones diferentes `IOException` y `NumberFormatException`. La primera hace referencia a las excepciones que se producen cuando un fallo interrumpe la ejecución de lectura o escritura de recursos (que puede ser trabajos con archivos externos a la aplicación o los archivos de la propia aplicación en si misma). La segunda hace referencia al error que se produce cuando falla la conversión de tipos de datos a números, por ejemplo, en este caso se intenta leer una línea del archivo y el valor obtenido convertirlo a un número que luego incrementa la variable `suma`. Si la línea leída no puede convertirse a un número se genera la excepción `NumberFormatException`.