TEMA 8. CLASES GENÉRICAS Y CONTROL DE EXCEPCIONES

INDICE

CONTROL DE EXCEPCIONES

El tratamiento de excepciones en Java es un mecanismo del lenguaje que permite gestionar errores y situaciones excepcionales. Debido a que el tratamiento de excepciones es uno de los pilares fundamentales del lenguaje, todo programador sabe cómo lanzarlas y capturarlas, pero (como cualquier aspecto fundamental en programación) es necesario conocer con cierta profundidad el propósito por el cual tenemos disponible dicho mecanismo, además de hacer un uso correcto de esta funcionalidad. En este artículo vamos a ver ambos aspectos: el qué y el cómo.

Una excepción en Java (así como en otros muchos lenguajes de programación) es un error o situación excepcional que se produce durante la ejecución de un programa. Algunos ejemplos de errores y situaciones excepcionales son:

- Leer un entero y que se introduzca el carácter 'l'
- Abrir un fichero que no existe
- Acceder al valor que está en la posición N de una colección que contiene menos de N elementos
- Enviar/recibir información por la red mientras se produce una pérdida de conectividad (problemas con sockets).

Todas las excepciones en Java se representan, a través de objetos que heredan, en última instancia, de la clase **java.lang.Throwable**.

TIPOS DE EXCEPCIONES

El lenguaje Java diferencia claramente entre tres tipos de excepciones: **errores**, **comprobadas** (en adelante **checked**) y **no comprobadas** (en adelante **unchecked**).

La clase principal de la cual heredan todas las excepciones Java es **Throwable**. De ella nacen dos ramas: **Error** y **Exception**.

La clase **Error** se utiliza para situaciones de una magnitud tal que una aplicación nunca debería intentar realizar nada con ellos (como errores de la JVM, desbordamientos de buffer, etc). Nosotros no la vamos a estudiar ya que nos vamos a centrar en las situaciones que sí se deben controlar.

La clase **Exception** se utiliza para aquellas situaciones que normalmente si solemos gestionar, y a las que comúnmente solemos llamar **excepciones**.

De la clase Exception nacen múltiples ramas: **ClassNotFoundException**, **IOException**, **ParseException**, **SQLException** y otras muchas, todas ellas de tipo checked. La única excepción (valga la redundancia) es **RuntimeException** que es de tipo unchecked y encabeza todas las de este tipo.

A pesar de que la diferencia entre las excepciones de tipo checked y unchecked es muy importante, es también a menudo uno de los aspectos menos entendidos dentro del tratamiento de excepciones. Veamos cada una de ellas con un poco más de detalle.

EXCEPCIONES COMPROBADAS. CHECKED

Una excepción de tipo checked representa un **error del cual técnicamente podemos recuperarnos**. Por ejemplo, una operación de lectura/escritura en disco puede fallar porque el fichero no exista, porque éste se encuentre bloqueado por otra aplicación, etc. Todos estas situaciones, además de ser inherentes al propósito del código que las lanza (lectura/escritura en disco) son totalmente ajenas al propio código, y deben ser (y de hecho son) declaradas y manejadas mediante excepciones de tipo checked y sus mecanismos de control.

En ciertas situaciones nuestro código no estará preparado para gestionar la situación de error, o simplemente no será su responsabilidad. En estos casos lo más razonable es relanzar la excepción y confiar en que un método superior en la cadena de llamadas sepa gestionarla.

Por tanto, todas las excepciones de tipo checked deben ser capturadas o relanzadas. En el primer caso, utilizamos el bloque **try-catch**.

Por ejemplo para controlar excepciones del tipo **IOException** cuando leemos desde System.in escribimos

```
import java.io.IOException;

public class IOExceptionMain {
 public static void main(String[] args) {
  try {
   byte[] nombre = new byte[30];
   System.out.println("Introduce tu nombre: ");
   System.in.read(nombre);
   System.out.println("Hola "+ new String(nombre));
  } catch (IOException ioe) {
   // muestro el error
   System.out.println("Error al leer los datos");
  }
 }
}
```

En caso de querer **relanzar la excepción**, debemos declarar dicha intención en la declaración del método que contiene las sentencias que lanzan la excepción mediante la cláusula **throws**.

Por ejemplo, para relanzar las excepciones del tipo **IOException** que se produzcan cuando escribamos en un fichero dentro del código del método **main** escribimos

```
import java.io.FileWriter;
import java.io.IOException;

public class Main {
   // En lugar de capturar una posible excepción, la relanzamos
   public static void main(String[] args) throws IOException {
      FileWriter fichero = new FileWriter("ruta");
      fichero.write("Esto se escribirá en el fichero");
   }
}
```

Hay que tener presente que cuando se relanza una excepción estamos forzando al código cliente de nuestro método a capturarla o relanzarla. Una excepción que sea relanzada una y otra vez hacia arriba terminará llegando al método primigenio y, en caso de no ser capturada por éste, producirá la finalización de su hilo de ejecución (**thread**).

Las preguntas que debemos hacernos son: ¿Cuándo debemos capturar una excepción? y, ¿Cuándo debemos relanzarla? La respuesta es muy simple.

Capturamos una excepción cuando tenemos que realizar algún tratamiento del propio error. Por ejemplo cuando

- Podemos recuperarnos del error y continuar con la ejecución
- Queremos registrar el error
- Queremos relanzar el error con un tipo de excepción distinto

Relanzamos una excepción cuando no es competencia nuestra ningún tratamiento de ningún tipo sobre el error que se ha producido.

EXCEPCIONES NO COMPROBADAS. UNCHECKED

Una excepción de tipo unchecked representa un **error de programación**. Uno de los ejemplos más típicos es el de intentar leer en un array de N elementos un elemento que se encuentra en una posición mayor que N:

```
int[] numerosPrimos = {1, 3, 5, 7, 9, 11, 13, 17, 19, 23};    // Array de diez elementos
int undecimoPrimo = numerosPrimos[10];    // Accedemos al undécimo elemento mediante el literal numérico 10
```

El código anterior accede a una posición inexistente dentro del array, y su ejecución lanzará la excepción unchecked del tipo **ArrayIndexOutOfBoundsException** (excepción de índice de array fuera de límite). Esto es claramente un error de programación, ya que el código debería haber comprobado el tamaño del array antes de intentar acceder a una posición concreta:

```
int[] numerosPrimos = {1, 3, 5, 7, 9, 11, 13, 17, 19, 23};
int indiceUndecimoPrimo = 10;

if(indiceUndecimoPrimo > numerosPrimos.length) {
   System.out.println("El índice proporcionado (" + indiceUndecimoPrimo + ") es mayor que el tamaño del array ("
+ numerosPrimos.length + ")");
} else {
   int undecimoPrimo = numerosPrimos[indiceUndecimoPrimo];
   // ...
}
```

El código anterior no sólo valida el tamaño de la colección antes de acceder a una posición concreta (el propósito fundamental del ejemplo), sino que evita el uso de literales numéricos asignando el índice del array a una variable.

El aspecto más destacado de **las excepciones de tipo unchecked** es que **no deben ser forzosamente declaradas ni capturadas** (en otras palabras, no son comprobadas), se controlan directamente en el código. Por ello **no son necesarios bloques try-catch** ni declarar formalmente en la firma del método el lanzamiento de excepciones de este tipo. Esto también afecta a métodos y/o clases más hacia arriba en la cadena invocante.

EXCEPCIONES CREADAS POR EL USUARIO

Aprovechando dos de las características más importantes de Java, la herencia y el polimorfismo, podemos crear nuestras propias excepciones de forma muy simple:

```
class MiException extends Exception {
   // ...
}
```

La clase del código anterior extiende a **Exception** y por tanto representa una excepción de tipo **checked**. Tal como su nombre indica, sería lanzada cuando durante una operación comercial no exista suficiente crédito.

Esta situación excepcional es inherente a nuestra transacción comercial y no se genera por un defecto de código (no es un error de programación), y por tanto debe gestionarse con anterioridad:

```
class CarritoDeLaCompra {
   // ...

   public void pagarCompraConTarjeta() {
      try {
         TarjetaDeCredito tarjetaPreferida = cliente.getTarjetaDeCreditoPreferida();
         tarjetaPreferida.realizarPago(getImporteCompra());
         cliente.enviarEmailConfirmación();
      } catch(CreditoInsuficienteException ciex) {
         // Informamos al usuario de crédito insuficiente
      }
   }
}
```

Si por el contrario deseamos crear una excepción de tipo **unchecked**, debemos hacer que nuestra clase extienda de **RuntimeException**. Volviendo al último ejemplo, podríamos pensar que CreditoInsuficienteException podría ser declarada como una excepción de tipo unchecked, ya que siempre es posible validar el saldo de la tarjeta de crédito con anterioridad al pago (como hacíamos con los índices del array).

Sin embargo esto no siempre es posible ni razonable ya que:

- No disponemos del código fuente, sólo somos clientes de una librería escrita por terceros
- Aunque dispusiéramos del código fuente, no deberíamos estar autorizados a conocer el crédito disponible de ningún cliente (esto es información muy sensible y por tanto, confidencial)

Antes de escribir tus propias excepciones, piensa detenidamente en que grupo encaja mejor su responsabilidad (checked o unchecked).

MALAS PRÁCTICAS DE USO DE EXCEPCIONES

Para convertirnos en maestros de las excepciones, debemos evitar el uso de aquellas malas prácticas que se han generalizado a los largo de los años (y por supuesto no inventar las nuestras…). La primera que vamos a ver es la más peligrosa y, a pesar de ello, también la más común:

```
try {
   // Código que declara lanzar excepciónes
} catch(Exception ex) {}
}
```

El código anterior ignorará cualquier excepción que se lance dentro del bloque try, o mejor dicho, capturará toda excepción lanzada dentro del bloque try pero la silenciará no haciendo nada (frustrando así el principal propósito de la gestión de excepciones checked: gestiónala o relánzala). Cualquier error de diseño, de programación o de funcionamiento en esas líneas de código pasará inadvertido tanto para el programador como para el usuario.

Lo correcto es por lo menos mostrar un mensaje informativo indicando la excepción que se ha producido.

Para mostrar el mensaje informativo podemos usar un objeto **logging** dentro de un bloque catch:

```
try {
   // Código que declara lanzar excepciónes
} catch(Exception e) {
   logging.log("Se ha producido el siguiente error: " + e.getMessage());
   logging.log("Se continua la ejecución");
}
```

También podemos usar el método informativo de Throwable **printStackTrace** que muestra una traza completa de la excepción:

```
try {
   // Código que declara lanzar excepciónes
} catch(Excepcion e) {
   e.printStackTrace();   // Podemos añadir cualquier tratamiento adicional antes y/o después de esta línea
}
```

Otro abuso del mecanismo de tratamiento de excepciones es cuando se está intentando **escribir código que mejore el rendimiento de la aplicación**:

```
try {
   int i = 0;
   while(true) {
      System.out.println(numerosPrimos[i++]);
   }
} catch(ArrayIndexOutOfBoundsException aioobe) {}
```

El ejemplo anterior itera nuestro fabuloso array de números primos sin preocuparse de los límites del array (tal como haría de manera formal un bucle for) hasta sobrepasar el índice máximo, momento en el cual se lanzará una excepción de tipo ArrayIndexOutOfBoundsException que será capturada y silenciada. Esto es un error porque:

- El tratamiento de excepciones está diseñado para gestionar excepciones y no para realizar optimizaciones.
- El código dentro de bloques try-catch no dispone de ciertas optimizaciones de las JVM más modernas (por ejemplo, y aplicable a nuestro caso, iteración de colecciones)
- El resultado es estéticamente horrible

Otro error común se produce cuando estamos creando nuestra propia librería de excepciones y nos excedemos declarando excepciones checked. Las excepciones checked son fabulosas ya que, al contrario que los códigos return de lenguajes como C, fuerzan al programador a manejar condiciones excepcionales, mejorando así la legibilidad del código. Sin embargo, esta obligación puede llegar a cargar el código cliente:

```
try {
   // Código que declara lanzar muchas excepciónes
} catch(UnTipoDeException ex1) {
   // Gestionar...
} catch(OtroTipoDeException ex2) {
   // Gestionar...
} catch(OtroTipoMasDeException ex3) {
   // Gestionar...
} catch(OtroTipoTodaviaMasDeException ex3) {
   // Gestionar...
```

```
}
```

El código anterior suele abrumar, y el cliente acabará tentado por la siguiente alternativa:

```
try {
   // Código que declara lanzar muchas excepciónes
} catch(Exception ex) {
   // Gestionar cualquier excepcion, pues todas heredan de Exception
   // Perdemos la ventaja de gestionar condiciones excepcionales concretas
}
```

Por ello, debes pensar detenidamente si tu excepción es de tipo checked o unchecked. Recuerda, cualquier situación excepcional que deje la aplicación en un estado irrecuperable y/o no sea inherente al propósito del código que la produce debe ser declarada como una excepción de tipo unchecked.

La siguiente mala práctica que vamos a ver está íntimamente relacionada con la anterior, y es la de lanzar excepciones de forma genérica:

```
public void miMetodo() throws Exception {
   // Código que declara lanzar muchas excepciones.
   // Sin embargo, en la firma del método declaramos lanzar una única super-clase de todas éllas
}
```

Nunca hagas lo anterior ya que es un absoluto ERROR. Los clientes de tu método no sabrán jamás con qué condiciones especiales se pueden encontrar, y por tanto no podrán gestionarlas; no tendrán más remedio que informar del error y detener la ejecución.

Por último existe una técnica para convertir toda excepción checked en unchecked:

```
public void noLanzoExcepcionesChecked() {
   try {
      // Código que lanza una o más excepciones de tipo checked
   } catch(Exception ex) {
      throw new RuntimeException("Se ha producido una excepción con el mensaje: " + ex.getMessage(), ex);
   }
}
```

El método del código anterior convierte cualquier excepción de tipo checked en una excepción de tipo unchecked, de manera que ningún cliente suyo esté forzado a declarar/gestionar ninguna de ellas.

En los últimos años ha crecido una comunidad de usuarios Java que abogan por eliminar el sistema de excepciones checked, que es justo lo que hace el código anterior. Si alguno de estos usuarios me lee, probablemente discrepe con mi idea de incluir dicho código (o cualquiera de sus variantes con menos ámbito de alcance y por tanto menos agresiva) dentro de lo que yo considero malas prácticas. Donde, y me incluyo en este grupo, unos vemos ventajas sobre el tratamiento de excepciones de tipo checked (porque nos da control sobre los errores que se producen a través de estructuras basadas en código legible y con un propósito claro), ellos ven desventajas (sobrecarga del lenguaje y en última instancia del código con una funcionalidad que, y en esto estoy de acuerdo, no es absolutamente perfecta). Existe actualmente un debate abierto sobre la verdadera utilidad de las excepciones checked. Tal vez en próximas versiones de Java (o en el próximo gran lenguaje orientado a objetos) se elimine cualquier concepto actual de error comprobado, pero en el momento actual no es así.

Personalmente considero el ejemplo anterior no un abuso del lenguaje sino un verdadero mal uso. La razón es todavía peor a la del penúltimo ejemplo (¡aquel que te dije no hacer nunca!): no sólo estamos aniquilando toda posibilidad de un tratamiento de excepciones que sea razonablemente útil, sino que lo más probable es que nuestra nueva y flamante excepción unchecked vaya subiendo hacia arriba en la cadena de llamadas y termine deteniendo el hilo de ejecución actual (puesto que, como ya sabes, éstas no tienen por qué ser gestionadas de forma obligatoria). A efectos prácticos, la única verdadera posibilidad de que sea capturada es que alguien haya declarado un bloque catch que declare capturar Exception (lo cual ya hemos dicho que es otro mal uso):

```
class MiClaseCliente() {
   public void miMetodoCliente() {
      try {
         obj.noLanzoExcepcionesChecked();
      } catch(Exception e) {
         // Bloque catch extremadamente genérico (mal uso)
         // Capturamos RuntimeException porque hereda de Exception (¿casualidad?)
         // El bloque try debe lanzar al menos una excepción checked (o este bloque es ilegal)
         // ¿Cuantos niveles estoy por encima del origen de la excepción convertida?
         // ¿Y si no existiera ningún bloque como este? Detención del thread (podría haberse evitado)
      }
   }
}
```

Existen ciertas situaciones en las que la conversión de una o más excepciones de tipo checked en unchecked es útil o práctica, pero son situaciones tan concretas y a menudo tan complejas y potencialmente peligrosas que no voy entrar en ningún tipo de detalle sobre éllas. Mi consejo final es: ¡NO CONVIERTAS EXCEPCIONES CHECKED EN UNCHECKED! (salvo que realmente sepas lo que haces).

RECOMENDACIONES DE USO DE EXCEPCIONES

Hay algunos principios de uso que debemos ver desde la perspectiva del haz esto, en lugar del no hagas esto.

Un buen uso del tratamiento de excepciones es **usar excepciones que ya existen**, en lugar de crear las tuyas propias, siempre que ambas fueran a cumplir el mismo cometido (que es básicamente informar y, en caso de las checked, obligar a gestionar). Se suelen usar excepciones que ya existen cuando se dispone de un profundo conocimiento del API que se está usando (en otras palabras, experiencia). Si un argumento pasado a uno de tus métodos no es del tipo esperado, o no tiene el formato correcto, lanza una excepción IllegarArgumentException en lugar de crear tu propia excepción. Esto es bueno porque:

- Uno de los pilares de Java es la reutilización de código (no reinventes la rueda)
- Tu código es más universal (FormatoInvalidoException puede no significar nada para un germanoparlante)

Otra recomendación que no suele llevarse a cabo nunca o casi nunca es la de lanzar excepciones acordes al nivel de abstracción en el que nos encontramos. Imaginemos una serie de clases que actúan como capas, una encima de otra (cuanto más arriba más abstracta, cuanto más abajo más concreta). Cuando se produce un error en las capas más bajas y éste se propaga hacia arriba, llega un momento en que dicho error representando una condición excepcional muy concreta se encuentra en un contexto muy abstracto. Esto tiene básicamente tres problemas: el primero, que puede ser importante, es que estamos contaminando el API de las capas superiores con suciedad de las inferiores. El segundo, que es importante, es que estamos desvelando detalles de nuestra implementación muchos niveles por encima de lo deseable. El último problema, que es importante y puede ser crítico, es que si en el futuro deseamos intercambiar una de las

capas más concretas y ésta ha cambiado su implementación, todas las capas por encima se romperán. Por tanto, debemos lanzar excepciones apropiadas a la abstracción en la que nos encontramos:

```
try {
   // Código que declara lanzar excepciones de bajo nivel
} catch(BajoNivelException bnex) {
   throw new AltoNivelException("Mensaje");
}
```

Por último, y para terminar, debes **documentar adecuadamente las excepciones** que lanza tu código. Para ello, detalla en tus Javadoc todas las excepciones que lanzan tus métodos, informando que condiciones van a provocar el lanzamiento de cada una de ellas:

```
/**
* @author David Marco
* @throws MiExcepcion se lanza en caso de producirse [...] condición especial.
*/
public class MiClase throws MiExcepcion {
   // ...
}
```

Los apuntes de excepciones los he adaptado de los apuntes de David Marco que se encuentran en la web http://www.davidmarco.es/articulo/tratamiento-de-excepciones-en-java.

ENTRADA DE DATOS DE TIPO NUMÉRICO DE FORMA ROBUSTA

A la hora de leer datos de tipo numérico por teclado mediante la clase Scanner, si introducimos valores erróneos por ejemplo al usar el '**.**' como separador decimal en vez de la '**,**', o al introducir el caracter 'l' en vez del número 1, se produce una excepción del tipo **java.util.InputMismatchException** que hace que la aplicación deje de funcionar.

La solución a este problema consiste en leer los datos de tipo numérico como datos de tipo String y después comprobar uno a uno los caracteres del String leído para ver si contiene un dato numérico con el formato que queramos. Para ello se pueden usar los métodos

***Integer**.parseInt(String s);*	Convierte un String a **int**
***Double**.parseDouble(String s);*	Convierte un String a **double**. El separador decimal es '**.**'
***Character**.isDigit(char caracter)*	Devuelve true si **caracter** es un dígito (0…9)
cadena.substring(int inicio, int fin);	Devuelve un String con los datos entre las posiciones inicio y fin

Si el String leído no contiene un valor numérico válido se lanza una excepción del tipo **NumberFormatException** que podemos controlar. En ese control de excepciones es dónde debemos realizar la conversión del String en un valor numérico válido.

EJERCICIOS DE CONTROL DE EXCEPCIONES

1. Realiza la clase Java LeerEnteroRobusto que lee un número entero de forma robusta (como un dato de tipo String) por teclado y lo muestra por pantalla. Comprueba el ejercicio convirtiendo los siguientes valores: carácter '**l' a 1,** caracteres '**O' y 'o' , a 0** cualquier otro carácter los ignora. Comprueba el ejercicio con los siguientes valores: **loO14a59 (1001459), 3,l41o9 (314109), lo 2O1(10201)**.

2. Realiza la clase Java LeerRealRobusto que lee un número real de forma robusta por teclado y lo muestra por pantalla. Comprueba el ejercicio con los siguientes valores: **3.14159 (3.14159), 3,14159 (3.14159)**, y

a3,14.15,9 (3.14159). El valor de separador decimal válido para un número real normal es ‘,’ pero tiene que ser ‘.’ si tenemos que usar *Double.parseDouble(String s)*. Los caracteres que no sean numéricos se eliminarán.

3. Realiza la clase Java ExceptionRacional que lee el numerador y el denominador de un número racional comprobando que sean números enteros válidos. Una vez que se sabe que son válidos se controla que el denominador no sea 0 usando una excepción del tipo **ArithmeticException**. **n=lo d=a1O4, n=lo d=o**

CLASES GENÉRICAS

Las clases genéricas son una funcionalidad que proporciona Java para poder crear clases que tengan un comportamiento similar independientemente del tipo de dato que usen como referencia.

Hemos utilizado clases genéricas cuando vimos las colecciones pues el comportamiento de las colecciones es el mismo independientemente del tipo de dato que contengan.

Las clases genéricas reciben también el nombre de plantillas ya que sirven de plantilla para la creación de distintas clases que diferirán únicamente en el tipo de dato.

Para crear una clase genérica debemos añadir **<T>** al final de la definición de la cabecera de la clase.

Para crear objetos de una clase genérica debemos indicar entre <> el tipo concreto del objeto que queremos crear, teniendo en cuenta que para los datos primitivos (int, double, ...) debemos usar sus clases contenedoras (Integer, Double, ...).

Por ejemplo, si queremos crear una clase Java de nombre ClaseGenerica que tiene un atributo de tipo genérico y de nombre valorgenerico que recibe un valor en el método constructor de la clase y que tiene un método de nombre escribir que muestra el contenido de valorgenerico por pantalla escribimos

```
public class ClaseGenerica<T> {
 T valorgenerico;
 ClaseGenerica(T valor){
  valorgenerico = valor;
 }
 @Override
 public String toString() {
  return (this.valorgenerico.toString());
 }
}
```

Para comprobar su correcto funcionamiento nos creamos otra clase de nombre ClaseGenericaMain. Para probar su funcionamiento con datos de tipo Integer, Double, y String escribimos

```
public class ClaseGenericaMain {
 public static void main(String[] args) {
  // Prueba el funcionamiento de ClaseGenerica
  int i=5;
  double d=3.14;
  String s="Hola Mundo";

  // Integer
  ClaseGenerica<Integer> gi= new ClaseGenerica<Integer>(i);
```

```
  System.out.println(gi);

  // Double
  ClaseGenerica<Double> gd= new ClaseGenerica<Double>(d);
  System.out.println(gd);

  // String
  ClaseGenerica<String> gs= new ClaseGenerica<String>(s);
  System.out.println(gs);
 }
}
```

MÉTODOS GENÉRICOS

En determinadas ocasiones podemos querer crear dentro de una clase un método que realice la misma función independientemente del tipo del objeto que reciba.

Por ejemplo, si tenemos la clase EscribirGenerico que contiene un método que se llama escribir y que muestra por pantalla el valor de un objeto que recibe como parámetro escribimos

```
public class EscribirGenerico {
 public <U> void escribir (U valor){
  System.out.println("El contenido de valor es " + valor);
 }
}
```

Para comprobar su correcto funcionamiento nos creamos otra clase de nombre EscribirGenericoMain. Para probar su funcionamiento con datos de tipo int, double, y String escribimos

```
public class EscribirGenericoMain {
 public static void main(String[] args) {
  // Prueba el funcionamiento de ClaseGenerica
  int i=5;
  double d=3.14;
  String s="Hola Mundo";

  // creo un objeto del tipo EscribirGenerico
  EscribirGenerico eg= new EscribirGenerico();

  // Integer
  eg.escribir(i);

  // Double
  eg.escribir(d);

  // String
  eg.escribir(s);

 }
}
```

EJERCICIOS DE CLASES Y MÉTODOS GENÉRICOS

4. Realiza la clase Java de nombre ClaseGenerica que tiene un atributo de tipo genérico y de nombre valorgenerico que recibe un valor en el método constructor de la clase y que tiene un método de nombre escribir que muestra el contenido de valorgenerico.

5. Realiza la clase Java de nombre ClaseGenericaMain para probar el funcionamiento de la clase ClaseGenerica.

6. Realiza la clase Java de nombre EscribirGenerico que contiene un método que se llama escribir y que muestra por pantalla el valor de un objeto que recibe como parámetro.

7. Realiza la clase Java de nombre EscribirGenericoMain para probar el funcionamiento de la clase EscribirGenerico.

8. Realiza la clase Java de nombre ContadorGenerico que tiene un atributo de tipo genérico y de nombre valor y un atributo de tipo int y de nombre contador.
   - Un constructor por defecto que pone la variable contador a 0.
   - Un constructor que permita dar valores a cada uno de los atributos de la clase.
   - Un constructor copia.
   - Setters y Getters necesarios para el correcto funcionamiento de la clase.
   - Método toString que devuelve un String con el formato "**Valor: " + valor + " Contador: " + contador**. Ej: "Valor: 2 Contador: 3"

9. Realiza la clase Java de nombre ArrayListContadorInteger que hace uso de la clase genérica ContadorGenerico para crear contadores para datos de tipo int. Dentro del método main crea cinco datos de prueba de tipo ContadorGenerico de Integer y los añade a un ArrayList del tipo correspondiente. Una vez añadidos los elementos al ArrayList, muestra el contenido del ArrayList por pantalla.

10. Realiza la clase Java de nombre ArrayListContadorString que hace uso de la clase genérica ContadorGenerico para crear contadores para datos de tipo String. Dentro del método main crea cinco datos de prueba de tipo ContadorGenerico de String y los añade a un ArrayList del tipo correspondiente. Una vez añadidos los elementos al ArrayList, muestra el contenido del ArrayList por pantalla.

11. Realiza la clase Java de nombre ArrayListContadorIntegerMenu que hace uso de la clase genérica ContadorGenerico para crear contadores para datos de tipo int. Dentro del método main muestra un menú por pantalla con las siguientes opciones
   - Añadir Entero. Pide un entero y lo añade al array. Si el valor ya existe incrementa su contador.
   - Buscar Entero. Pide un entero y lo busca en el array. Devuelve la posición en que se encuentra o -1 si no se encuentra en el array.
   - Borrar Entero. Pide un String y lo elimina, si es que existe, del array.
   - Listar Array. Muestra todos los elementos del array por pantalla, si es que tiene elementos.
   - Salir. Realiza las operaciones necesarias para la correcta finalización del programa.

12. Realiza la clase Java de nombre ArrayListContadorStringMenu que hace uso de la clase genérica ContadorGenerico para crear contadores para datos de tipo String. Dentro del método main muestra un menú por pantalla con las siguientes opciones
   - Añadir String. Pide un String y lo añade al array. Si el valor ya existe incrementa su contador.
   - Buscar String. Pide un String y lo busca en el array. Devuelve la posición en que se encuentra o -1 si no se encuentra en el array.
   - Borrar String. Pide un String y lo elimina, si es que existe, del array.
   - Listar Array. Muestra todos los elementos del array por pantalla, si es que tiene elementos.
   - Salir. Realiza las operaciones necesarias para la correcta finalización del programa.

GENERACIÓN DE NÚMEROS ALEATORIOS

En determinadas ocasiones necesitamos usar números aleatorios. La clase Java **Random** proporciona un generador de números aleatorios.

Para indicar a nuestro programa que vamos a usar números aleatorios debemos importar la clase Java **java.util.Random**.

Una vez importada la clase, debemos crear un objeto de la clase Random para poder empezar a generar números aleatorios.

La clase Random dispone de métodos que permiten generar datos aleatorios de distinto tipo.

- **nextInt**(). Genera un número aleatorio entero de tipo int con un valor entre 2^{-32} y 2^{32}
- **nextLong**(). Genera un número aleatorio entero de tipo long con un valor entre 2^{-64} y 2^{64}
- **nextFloat**(). Genera un número aleatorio real de tipo float con un valor entre 0 y 1.
- **nextDouble**().Genera un número aleatorio real de tipo double con un valor entre 0 y 1.

Lo más habitual, para generar un número aleatorio, es usar la función nextDouble() aunque devuelva un valor entre 0 y 1. El truco consiste en multiplicar el resultado por el valor máximo del rango de números que queremos generar teniendo en cuenta que los valores generados irán desde 0 hasta ese número.

Por ejemplo, si queremos generar un número entero aleatorio entre 0 y 10 escribimos

```
Random rnd = new Random();
int numero = (int)(rnd.nextDouble()*10.0);
```

Si el rango de número no incluye el 0 y empieza en el número 1 debemos sumar 1 al número generado. Por ejemplo, si queremos generar un número entero aleatorio entre 1 y 10 escribimos

```
Random rnd = new Random();
int numero = (int)(rnd.nextDouble()*9.0)+1;
```

En determinadas situaciones los valores aleatorios pueden ser muy predecibles o repetirse con mucha frecuencia. Para mejorar la generación de números aleatorios podemos cambiar la semilla que es el valor que se usa como base para la generación de números aleatorios.

El método que se usa para cambiar la semilla es **setSeed**().

Por defecto Java usa como semilla el valor devuelvo por el método **System.currentTimeMillis**() que es el número de milisegundos que contiene la fecha actual del sistema. Al ser un dato que cambia constantemente es un buen valor para la generación de números aleatorios. Si no indicamos ninguna semilla es como si indicáramos lo siguiente

```
rnd.setSeed(System.currentTimeMillis());
```

Este método de generación de números aleatorios hace que algunos valores sean difíciles de alcanzar. Por ejemplo si en los ejemplos anteriores pusiéramos como condición de fin que el número valga 10 nos daríamos cuenta de que ese valor es prácticamente imposible que lo alcance.

Para solucionar esto se multiplica el valor generado por un valor mucho más grande y se realiza una división entera por el valor máximo del rango que queramos generar.

Por ejemplo, para generar un número entre 1 y 10 asegurándonos de que el valor 10 aparece alguna vez escribimos

```
numero = ((int)(rnd.nextDouble()*100000.0))%10 + 1;
```

EJERCICIOS DE NÚMEROS ALEATORIOS

13. Realiza la clase Java de nombre Random010 que genera un número aleatorio entero entre 0 y 10 y lo muestra por pantalla.

14. Realiza la clase Java de nombre Random110 que genera un número aleatorio entero entre 1 y 10 y lo muestra por pantalla.

15. Realiza la clase Java de nombre Random110Seed que usa como semilla la fecha del sistema en milisegundos y genera números aleatorios enteros entre 1 y 10 y los muestra por pantalla hasta que se genera el número 10.

16. Realiza la clase Java de nombre Random110Fin10 que genera números aleatorios enteros entre 1 y 10 y los muestra por pantalla hasta que se genera el número 10. La generación se hace multiplicando por 100000.0 el valor generado y haciendo después su división entera.

17. Realiza la clase Java de nombre RandomNumeroSecreto que genera un número aleatorio entero entre 1 y 10 que el usuario deberá adivinar. Después de cada intento mostrará un mensaje indicando si lo ha acertado o si el número que ha metido es menor o mayor que el número secreto. La generación del número aleatorio se hace multiplicando por 100000.0 el valor generado y haciendo después su división entera.

18. Realiza la clase Java de nombre RandomMasterMind que genera número aleatorio de cuatro cifras que no se repiten. Después de cada intento muestra un mensaje indicando si ha acertado el número o, en caso contario, el número de cifras correctas que hay y el número de cifras correctas que estan en la posicion correcta. La generación del número aleatorio se hace multiplicando por 100000.0 el valor generado y haciendo después su división entera.