# Java SE

Curso Avanzado de Java Standard Edition

*Arturo Rivas Rojas*

*30 de diciembre de 2018*

# Índice general

# CAPÍTULO 1

# Polimorfismo

Posibilidad de construir un método, con comportamiento diferente según la clase a la que pertenece, cuando las clases están relacionadas ya sea por herencia, interfaces o clases abstractas.

## 1.1. Herencia

En la herencia el polimorfismo se lleva a cabo mediante la sobreescritura de métodos. Por ejemplo en el constructor normalmnete una subclase tiene los mismos atributos que la clase padre y algunos otros, por lo que su constructor debe ser diferente entonces usando la palabra super llamamos a la clase padre y podemos lalmar a su cosntructor reciclandolo y agregar lo que falte.

## 1.2. Interfaces

Solo pueden contener costantes y definición de métodos, es decir solo la firma del método si implementación. Siempre que se implementa una interfaz se esta realizando polimorfismo.

## 1.3. Clases Abstractas

En una clase abstracta no se imlementan todos los métodos y no se pueden crear instancias de estas clases.

```
public abstract class Figura{
    abstract void dibujate();
}
```

La palabra **abstract** es la clave para las clases abstractas y los métodos abstractos son los métodos que son obligatorios de implementar.

```
class Triangulo extends Figura{
    abstract void dibujate();
}
```

Los métodos abstractos se heredan y se puede decidir si implementar el método o conservar sus abstracción.

CAPÍTULO 2

# JavaDoc

Genera documentación en HTML desde el código Java.

## 2.1. Funcionamiento

Java genera la documentación a través de los comentarios.

```
// un comentario común de una sola linea.
/* comentario común
de multiples lineas */
/** comentario de documentación
* <h1> Titulo </h1>
* <p> saltos de linea
* */
```

solo los comentarios dentro del tercer tipo son tomados en cuenta para la generación del javadoc.

Dentro de los comentarios javadoc se pueden utilizar etiquetas de HTML para dar formato a nuestra salida.

## 2.2. Etiquetas

- **@author** : autor.
- **@version** : versión del código.
- **@since** : año, fecha.
- **@see** : sugerencia de visualización, como más información.
- **@param** : parámetros.
- **@return** : retorno del método.
- **{@link clase/interfaz}** : genera un link a la clase o interfaz.
- **{inheritDoc}** : historia del método, herencia o implementación de clases.

- **{code código}** : da formato al texto código.
- **@exception** or **@throws** : exceptiones que puede lanzar.
- **{@value #constant}** : muestra el valor de la constante.

# CAPÍTULO 3

# Clases Anidadas

Las clases anidadas, o nested classes. Nacen por la necesidad de tener agrupadas clases que no pueden o deben existir por separado.

Una clase anidada tiene acceso a los miembros (atributos y métodos) de su clase contenedora, incluidos los privados. Lo inverso no es verdadero.

## 3.1. Clasificación

- Static Nested Classes

  Se declaran utilizando la palabra reservada **static**. Y se diferencian por tener la limitación de no poder acceder a los miembros no staticos de su clase contenedora. Ademas de que no es necesario instancia a la clase contenedora para instanciar a una **static nested class**.

```
// clase contenedora
public class Outer{
    // clase anidada estática
    public static StaticInner{
        // método de la clase anidada
        public run(){
        }
    }
    // método main de la clase contenedora
    public static void main (String[] args){
        // instanciando a la clase anidada
        Outer.StaticInner nested =  new     Outer.StaticInner();
        // llamando a un método de la clase anidada.
        nested.run():
        // el método run() no es estático por lo que    fue necesario instaciar a la clase para
    }
}
```

- Inner classes

  Son clases que se declaran de la manera común, y se diferencia por que para instanciarlas es necesario intanciar a la clase contenedora. Son mucho mas versátiles que las estáticas ya que pueden llamar a todos los miembros de la clase que los contiene.

```
// clase contenedpra
public class Outer {
    //declaración de la clase interna
    public class Inner {
        //atributos y métodos de la clase anidada
    }
    // método main de la clase contenedora
    public static void main(String[] args) {
        // instanciando de la clase contenedora
        Outer outer = new Outer():
        // instanciando de la clase anidada
        outer.Inner inner = outer.new Inner();
    }
}
```

  - Method Local Inner Classes

    Es un subconjunto de las Inner Classes pues su unica diferencia es que solo existen dentro de uno de los métodos de la clase contenedora.

```
// clase contenedora
public class Outer {
    // método run() de la clase contenedora
    void run(){
        // declaración de la clase local de     método anidada
        public class Local {
            // método run de la clase anidada
            void run() {
                //implementación
            }
        }
        // instancia de la clase anidada
        Local local = new Local()
        // llamada al método de la clase    anidada
        local.run();
    }
    // método main de la clase contenedora
    public static void main(String[] args)
    {
        // instacia de la clase contenedora
        Outer outer = new Outer();
        // llamada al método run() de la    clase contenedora
        outer.run();
    }
}
```

  - Anonymous Inner Classes

    Son clases que nos permiten instanciar clases abstracta implementando los métodos abstractos de la clase abstracta al mismo tiempo en el que instanciamos el objeto.

```
// Abstracta.java
//clase abstracta
public abstract Abstracta {
 public abstract void run();
}
// Anonima.java
public class Outer {
    public void ejecutar() {
    // instanciando clase anónima
    Abstracta abstracta = new Abstracta() {
        void run() {
            // implementación
        }
    }
    // llamando método de la clase anónima
    abstracta.run();
    }
}
```

# CAPÍTULO 4

## Enumeration

Un enumerado o (Enum) es una clase “especial” que limitan la creación de objetos a los explícitamente en la implementación de la clase. La única limitación que tienen los enumerados respecto a una clase normal es que este debe ser privado para que no se puedan crear nuevos objetos.

```
public enum Day {
    // cosntructores según la constante elegida
    SUNDAY("Domingo"),
    MONDAY("Lunes"),
    TUESDAY("Martes"),
    WEDNESDAY("Miercoles"),
    THURSDAY("Jueves"),
    FRIDAY("Viernes"),
    SATURDAY("Sabado")
    // atributo de la clase Day
    private String spanish;
    // constructor de la clase Day
    private Day(String s) {
        spanish = s;
    }
    // método de la clase Day
    public String getSpanish() {
        return spanish;
    }
}

System.out.println(Day.Friday); // imprime Friday
System.out.println(Day.Monday.getSpanish(); // imprime Lunes
```

# CAPÍTULO 5

# Interfaces

Solo se pueden componer de métodos sin implementación y constantes. Por lo mismo no se puende instanciar una interfaz.

Apartir de java 8 se pueden implementar métodos con modificador de acceso default y desde java 9 pueden implementarse métodos con el modificador de acceso private.

```
public interface MyInterface {
    default void defaultMethod() {
        privateMethod("Hola Mundo desde el default");
    private void privateMethod(String s) {
        System.out.println(s);
    void commonInterfaceMethod();
```

Los métodos privados no solo podran ser accedidos desde la misma interfaz en el ejemplo se hace a través del metodo default.

## 5.1. Data Access Object (DAO)

Es un patron de diseño orientado a interfaces en donde la interfaz implementa los metodos CRUD.

CAPÍTULO 6

# Interfaces vs Clases Abstractas

Ambas tienen las mismas capacidades en cuestion de implementación, pero las clases abstractas se ven limitadas por la herencia.

Las interfaces son mucho mas flexibles y pueden ser ocupadas por cualquier clase y pueden implemnetarse más de una interfaz.

Se recomiendo programar orientado a interfaces.

# CAPÍTULO 7

# Manejo de Errores

Todo error que pueda ocurrir en Java se genera un mensaje Throwable.

## 7.1. Clasificación

- Throwable
  - Error: son errores que ocurren en la JVM y son ajenas al programador
  - Exception: son los errores que son previsibles y manipulables por el programador
    - Runtime: solo son visibles durante la ejeccución.
    - Otras: son todas las excepciones que pueden surgir por la interacción con un componente externo.

## 7.2. Manejo

El manejo de errores ocnsiste en añadir un bloque de código para definir el comportamiento de la aplicación cuando ocurra un error.

### 7.2.1. Try-catch-finally

```
try{
    // código vulnerable que puede lanzar excepciones del tipo 1 o 2
} catch (ExceptionType1 name) {
    // código que se ejecutará si se lanza una excepción del tipo 1
} catch (ExceptionType2 name) {
    // código que se ejecutará si se lanza una excepción del tipo 2
} finally {
    // código que siempre se ejecuta
}
```

### 7.2.2. Try-with-resources

```
BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
try(reader) {
    // código que puede lanzar excepciones
} catch (Exception e) {
    // manejo de errores
}
```

El try-with-resources se encarcaga de cerrar el recurso del try, asi nos evita tener que escribir el bloque finally y los try-catch anidados correspondientes.

# CAPÍTULO 8

# Java Data Base Connectivity (JDBC)

Es un conjunto de API para conexiones con la base de datos.

Cada Motor de Base de Datos necesita una biblioteca jar distinta que llamada JDBC driver.

## 8.1. Componentes

- **DriverManager** nos permite generar el objeto para conectar con la base de datos.
- **Connection** nos genera una sesion durante la conexion a la base de datos.
- **Statement** nos ayuda a generar las consultas.
- **PreparedStatement** se diferencia de statement pues puede recibir paramentros para la clausula WHERE.
- **ResultSet** es un interfaz para manipular los datos obtenidos de la base de datos. Manejando cada columna como un objeto.

## 8.2. Metodos CRUD

| Método | Devuelve | Uso |
|---|---|---|
| **executeQuery**(String sqlString) | Considencias **ResulSet** | Consultas **SELECT** |
| **executeUpdate**(String sqlString) | Filas afectadas **int** | Modificaciónes **INSERT,UPDATE, DELETE, o DDL** |
| **execute**(String sqlString) | **boolean** | Cualquioper comando |

Cuadro 8.1: Métodos CRUD

## 8.3. Ejemplos

- Conexión a BD de mysql

```
// paquete de la interfaz
package interfaces;
// clases que ocupamos importar
import java.sql.Connection;
import java.sql.DriverManager;

public interface DBConnection {
    // metodo que podemos llamar desde cualquier clase que iplemnete a la interfaz DBConection
    default Connection conectar() {
        Connection conexion = null;
        try {
            // importar clase mediante el nombre.
            Class.forName("com.mysql.jdbc.Driver");
            //jdbc:MotorBD:dirección y puerto/BD, user, password
            conexion = DriverManager.getConnection("jdbc:mysql://localhost:3306/tt","root","root");
        } catch (Exception e) {
            e.printStackTrace();
        }
        return conexion;
    }
}
```

- Consulta a la BD

```
String titulo;
// try-with-resources con la conexion por si hay alguna excepción atraparla y cerrar la conexión al fin
try(Connection conexion = conectar()) {
    // Query que queremos ejecutar
    String query = "SELECT * from tt" +
    "WHERE titulo = ?" + // cada '?' sera un parametro en el preparedStatement que podemos modificar co
    "AND autor = ?";
    // Cargamos el query
    PreparedStatement preparedStatement = conexion.preparedStatement(query);
    // Definimos los paramateros es decir los '?'
    preparedStatement.setString(1,"Inteligencia Artificial"); // la numeración inicia desde el 1
    preparedStatement.setString(2,"Juan Pérez");
    // ejecutamos el query y guardamos la respuesta
    ResultSet consulta = preparedStatement.executeQuery();
    // Mientras haya un siguiente resultado
    while(consulta.next()) {
        // guardamos el valor de la columan "titulo" en una string.
        titulo = consulta.getString("titulo");
        System.out.println("")
    }
}
```

- Modificaciónes a la BD

```
// try-with-resources con la conexion por si hay alguna excepción atraparla y cerrar la conexión al fin
try(Connection conexion = conectar()) {
    // objeto Statement que executara el query
    Statement statement = conexion.creatStatement();
    // Query que queremos ejecutar
    String query = "INSERT INTO tt(" + titulo + ", " + autor + ", " + sinodales");";
    // si se ingresaron 1 o mas registros
    if(statement.executeUpdate(query)>0) {
        Se imprime el mensaje
        System.out.println("registro exitoso.")
    }
}
```

# CAPÍTULO 9

## Interfaces Funcionales

Consiste en un único método abstracto SAM (Single Abstract Method), su anotación es @FuntionalInterface.

Ejemplo:

```
@FuntionalInterface
public interface Greeting {
    public void perform();
}
```

# CAPÍTULO 10

# Programación Funcional

Paradigma de programación funcional, un paradigma son uan seria de normas que un lenguaje se ajusta para dar solución a un problema.

## 10.1. Clasifiación de paradigmas

- Programación
  - Imperativa : Se enfoca en el “Cómo” se quiere lograr.
    - Estructurada : Basic
    - Procedimental : Go , C
    - OOP : Java , PHP
  - Declarativa : Se basa en el “Qué” se quiere lograr.
    - Funcional : Haskell , Elixir, Scala, Erlang
    - Lógico : Lisp , Prolog.

## 10.2. Funciones de orden superior

Una función es una operación que recibe un dato lo procesa y retorna otro data. Pero para distinguir a la programación funcional es porque existen **funciones de orden superior**, funciones que reciben y retornan funciones.

# CAPÍTULO 11

# Lambdas

El formato de una lambda en Java y en la mayoria de los lenguaje es el siguiente:

```
(parámetros)->{cuerpo-lambda}
```

Una lambda se debe utilizar cuando hay fragmentos de código con tiempos de vida muy cortos. Encapsulan código especifico.

Para implementar una Lambda es necesario el uso de una Interfaz funcional o **clases anónimas**.

## 11.1. Implementación

```java
// HolaMundoInterfaz.java
@FuntionalInterface
public interface HolaMundoInterfaz {
    public void holaMundo(String nombre);
}
// HolaMundoClase.java
public class HolaMundoClase {
    public static void main(String[] args) {
        // Hacemos una instacia de la interfaz y utilizando una lambda definimos el método SAM de la in
        HolaMundoInterfaz objetoHolaMundo = (String nombre) -> {
            System.out.println("Hola Mundo" + nombre);
        };
        // utilizamos el metodo que acabamos de definir
        objetoHolaMundo.imprimir("Arturo"); // imprime "Hola Mundo Arturo"
    }
}
```

## 11.2. Como variables

Retomando el ejemplo anterior vemos que podemos ver que se puede simplificar aun mas la delcaraciond de una lambda.

```
HolaMundoInterfaz objetoHolaMundo = nombre -> System.out.println("Hola Mundo" + nombre);
```

## 11.3. Recursividad

La programación funcional no utiliza iteraciones (for o whiles) si no que se basa en la recursividad. Por lo tanto en las lambdas no hay bucles.

El metodo colección.**forEach(lambda)** de las colecciones de datos trabaja de manera recursiva. Seria el equivalente de un map en haskell.

```
ArrayList<Film> = new ArrayList();
films.forEach(f -> System.out.println(f.toString()));
// estas dos lineas hacen exactamente lo mismo
films.forEach(System.out::println);
```

El metodo colección.**stream()** es un metodo que habilita a las colecciones trabajar con lambdas.
Y colección.stream().**filter(lambda)** es un metodo que nos permite filtrar una colección según una lambda. El metodo **filter()** retorno un **stream()** por lo que se pueden concatenar multiples **filter()**.

```
// declaramos una lista que es una colección
List<String> lista = Arrays.asList("Hola","Mundo","","?");
// filtramos la lista ignorando las cadenas vacias. E imprimimos la lista resultante
lista.stream().filter(f -> !f.isEmpty()).forEach(System.out::println);
// imprime "Hola\n" "Mundo\n" "?\n"
```

## 11.4. Interfaces Predicate y Consumer

```
// declaración del predicado
Predicate<clase> predicado =  l -> ...;
// filtramos por el predicado
lista.stream().filter(predicado);
// declaramos un consumidor
Consumer<clase> consumidor =  l -> ...;
// se realiza por cada elemento de la lista el consumidor.
lista.forEach(consumidor);
```

La interfaz **Predicate** nos permite generar los predicados del método **filter()** de manera separada y así tener un código mas legible.

La interfaz **Consumer** nos permite generar operaciones para el método **forEach()** mas complejos.