



Módulo Profesional: Big Data Aplicado

Scala

case class.

```
// Definición de una case class
case class Persona(nombre: String, edad: Int)

// Uso de la case class
object EjemploCaseClass extends App {
    // Crear una instancia
    val persona1 = Persona("Juan", 25)
    // Acceder a los campos
    println(s"Nombre: ${persona1.nombre}, Edad: ${persona1.edad}")
    // Crear una copia modificando uno de los campos
    val persona2 = persona1.copy(edad = 30)
    println(s"Nombre: ${persona2.nombre}, Edad: ${persona2.edad}")
    // Comparación (case classes implementan `equals` y `hashCode`)
    println(persona1 == persona2) // false
    // Representación como cadena (toString implementado automáticamente)
    println(persona1) // Persona(Juan,25)
}
```

Case Classes. Propiedades. Diferencias con class

Inmutabilidad

case class	class
Las propiedades declaradas como parámetros en una case class son inmutables (implícitamente val)	Las propiedades deben declararse explícitamente como val o var para definir su mutabilidad.

Case Classes. Propiedades. Diferencias con class

Creación de Instancias

case class	class
No requiere la palabra clave <i>new</i> .	Requiere la palabra clave <i>new</i> .

Case Classes. Propiedades. Diferencias con class

Métodos generados

Case class	class
Genera automáticamente métodos como <code>ToString</code> , <code>equals</code> , <code>copy</code>	No genera estos métodos automáticamente. Se deben implementar manualmente si son necesarios.

Case Classes. Propiedades. Diferencias con class

Pattern Matching o Coincidencia de patrones

case class	class
Es compatible con el pattern matching de forma predeterminada	No es compatible con el pattern matching a menos que se implemente manualmente el método <i>(método para extraer información de los objetos)</i>

El Pattern Matching evalúa una expresión y la compara con una serie de patrones predefinidos. Cuando encuentra un patrón que coincide, ejecuta el bloque de código asociado a ese patrón.

Case Classes. Propiedades. Diferencias con class

Pattern Matching o Coincidencia de patrones

// Usar pattern matching para determinar el mensaje basado en la edad

```
persona match {  
    case Persona(_, edad) if edad < 18 => println("Es un menor de edad.")  
    case Persona(_, edad) if edad >= 18 && edad <= 65 => println("Es un adulto.")  
    case Persona(_, edad) if edad > 65 => println("Es una persona mayor.")  
}
```

* case Persona (_, edad) => el _ indica que se está ignorando el campo nombre de la instancia Persona. Es decir, no nos interesa el valor del nombre en este patrón, solo nos interesa el valor de edad

Case Classes. Propiedades. Diferencias con class

Comparación

Case class	class
La comparación es estructural. Compara los valores de los campos.	La comparación es referencial por defecto. Compara referencias de memoria

Case Classes. Propiedades. Diferencias con class

Clonado

Case class	class
Proporciona el método <i>copy</i> para crear copias con valores modificados	No tiene un método <i>copy</i> . Se debe implementar manualmente

Case Classes. Propiedades. Diferencias con class

Serialización

Case class	class
Las case class son serializables por defecto	No son serializables automáticamente. Se debe implementar manualmente

La serialización es el proceso de convertir un objeto en una representación que pueda ser almacenada o transmitida. Ejemplos, guardar un objeto en un archivo; enviar un objeto a través de la red; almacenar un objeto en caché.

Case Classes. Propiedades. Diferencias con class

Herencia

Case class	class
No puede extender directamente otra case class pero puede extender una clase o trait.	Puede heredar de cualquier clase o trait.

Case Classes. Propiedades. Diferencias con class

Uso de colecciones

Case class	class
Ideal para usar como claves en mapas (Map) o en estructuras que requieran hashcodes confiables.	No recomendado para este uso sin sobreescribir explícitamente equals y hashCode.

Estructuras hashCode es un valor numérico generado a partir de un objeto utilizando una función hash. Este código representa de manera única el contenido del objeto. Ejemplo:
val str = "Scala"
Println (str.hashCode()) → resultado: 83835195

¿Cuándo usar case class?

- 1. Necesitas comparaciones por valor (no por referencia)**
- 2. Necesitas las instancias inmutables**
- 3. Quieres desestructuración fácil (Pattern Matching)**
- 4. Necesitas crear instancias de manera concisa**
- 5. Generación automática de `toString`, `copy`, `hashCode`**

Case Classes

1. Los parámetros de la clase son campos

```
case class Persona (nombre: String, edad: Int)
//1. Los parámetros de la clase son campos
val Roberto = new Persona ("Roberto", 25)
println (Roberto.nombre)
```

Case Classes

2. sensible `toString`

```
//2. sensible toString  
println(Roberto.toString)
```

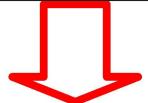


```
Persona(Roberto,25)
```

Case Classes

3. equals and hashCode

```
//3. equals and hashCode  
val Roberto2 = new Persona ("Roberto", 25)  
println (Roberto == Roberto2)
```

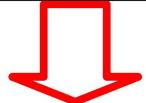


¿Verdadero o Falso?

Case Classes

3. equals and hashCode

```
//3. equals and hashCode  
val Roberto2 = new Persona ("Roberto", 25)  
println (Roberto == Roberto2)
```

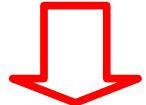


Verdadero (True)

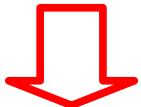
Case Classes

4. Case Classes se puede copiar

```
//4. CCs se puede copiar métodos  
val Roberto3 = Roberto.copy(edad = 45)  
println (Roberto3)
```



En este caso Roberto3, tiene el mismo nombre pero la edad es distinta



```
Persona(Roberto,45)
```

Case Classes. Ejercicio 1

```
/*Ejercicio 1: Utiliza case class  
Un libro tiene las siguientes propiedades:  
- Título: Nombre del libro de tipo cadena.  
- Autor: Nombre del autor de tipo cadena.  
- Año de publicación: El año de tipo entero.
```

Se debe realizar las siguientes acciones:

- Crea un libro con los siguientes valores:
 - Título: "1984"
 - Autor: Anónimo
 - Año: 1986
- Imprimir por pantalla los datos del libro

```
*/
```

Case Classes. Ejercicio 2

```
/*Ejercicio 2
Representar diferentes tipos de operaciones matemáticas: Suma, Resta y Multiplicación.
Cada operación tendrá dos operandos (a y b) y utilizamos pattern matching para realizar
la operación correspondiente y devolver el resultado
*/
```

Enums

```
object Enums {
    enum Permisos {
        case READ, WRITE, EXECUTE, NONE
        //añadir métodos/campos
        def abrirDocumento(): Unit =
            if (this == READ) println("abrir documento...")
            else println ("Lectura no permitida")
    }
    val algunosPermisos: Permisos = Permisos.READ
    def main (args: Array[String]): Unit = {
        algunosPermisos.abrirDocumento()
    }
}
```

Enums. Ejercicio

```
/* Ejercicio 1: Colores primarios
 - Representa los colores primarios: Rojo, azul y amarillo.
 - Imprime por pantalla todos los colores primarios
 */
// Definimos un enum para representar los colores primarios
```

Exceptions

```
val x:String = null  
println (x.length)
```

```
Exception in thread "main" java.lang.ExceptionInInitializerError  
at Exceptions.main(13_Exceptions.scala)
```

```
Caused by: java.lang.NullPointerException: Cannot invoke "String.length()" because the return value of "Exceptions$.x()" is null  
at Exceptions$.<clinit>(13_Exceptions.scala:4) Exceptions.scala:4
```

Exceptions

```
//Excepciones
def obtenerEntero(conException: Boolean): Int =
    if (conException) throw new RuntimeException("No es un entero")
    else 24

try {
    //código que debe ser capturada la excepción
    obtenerEntero (true)
} catch {
    case e: RuntimeException => println ("capturado Runtime exception")
} finally {
    //código que se ejecutará
    println("finally")
}
```

Exceptions: ¿Qué pasa?

```
//Excepciones
def obtenerEntero(conException: Boolean): Int =
    if (conException) throw new RuntimeException("No es un entero")
    else 24

try {
    //código que debe ser capturada la excepción
    obtenerEntero (true)
} catch {
    case e: NullPointerException => println ("capturado Runtime exception")
} finally {
    //código que se ejecutará
    println("finally")
}
```

Exceptions. Define tus propias excepciones

```
val falloPotencial = try {  
    //código que debe ser capturada la excepción  
    obtenerE+...  
} catch { type RuntimeException: RuntimeException  
    case e: RuntimeException => println ("capturado Runtime exception")  
} finally {  
    //código que se ejecutará  
    // es opcional  
    // no influye en el resultado de la expresión  
    //usa finally sólo si tiene algún efecto  
    println("finally")  
}  
  
println(falloPotencial)
```

Exceptions. Ejercicio 1

```
/* Ejercicios
1. Calculadora
    - suma (x,y)
    - restar (x,y)
    - multiplicar (x,y)
    - dividir (x,y)

    Throw
        - OverflowException si suma (x,y) excede int.MAX_VALUE
        - UnderflowException si la resta (x,y) excede int.MIN_VALUE
        - MathCalculationException si la división por 0
*/
```

Exceptions. Ejercicio 2

```
/*Ejercicio 2: Manejo de Excepciones con Conversión de tipos  
Función para convertir una cadena a un tipo entero.  
Si la conversión falla, se debe lanzar una excepción NumberFormatException  
que será capturada y manejada adecuadamente  
*/
```

