Módulo Profesional: Big Data Aplicado

Scala

Tipos de datos Abstractos y clases

```
//abstract
abstract class Animal {
  val TipoCriatura : String
  def comer: Unit
}

val animal  = new Animal
```

Una clase abstracta no puede ser instanciada porque contiene atributos y /o métodos que no están implementados.
Para crear un objeto de una clase abstracta se debe primero crear una clase hija que sobreescriba e implemente todos los métodos y atributos.

Una clase abstracta sirve como una plantilla o modelo base que otras clases pueden heredar e implementar. Por ejemplo, puede ser útil definir una estructura común para un conjunto de clases relacionadas pero necesitas que cada clase hija implemente métodos o atributos.

Tipos de datos Abstractos y clases

```
//abstract
abstract class Animal {
  val TipoCriatura : String
  def comer: Unit
}

class Perro extends Animal {
    ↑ TipoCriatura
  override val TipoCriatura: String = "Doméstico"
   ↑ comer
  def comer: Unit = println ("comer perro")
}
```

Rasgos (Traits)

```scala
//abstract
abstract class Animal {
  val TipoCriatura : String
  def comer: Unit
}
```

1

```scala
class Perro extends Animal {
  ↑ TipoCriatura
  override val TipoCriatura: String = "Perro"
  ↑ comer
  def comer: Unit = println ("comer perro")
}
```

2

Rasgos (Traits)

```scala
//traits (rasgos)
trait Carnivoro {
  def comer (animal: Animal): Unit
}
```

3

```scala
class Cocodrilo extends Animal with Carnivoro {
  ↑ TipoCriatura
  val TipoCriatura: String = "croc"
  ↑ comer
  def comer: Unit = println("ñam ñam")
  ↑ comer
  def comer(animal: Animal): Unit =
      println( s"Yo sou un cocodrilo y me como ${animal.TipoCriatura}")
}
```

4

Rasgos (Traits)

```
val perro = new Perro
val croc = new Cocodrilo
croc.comer (perro)
```

5

Características:

1.- Pueden contener métodos abstractos y concretos
2.- No pueden tener parámetros en el constructor
3.- Una clase puede heredar de la clase padre y de un trait o más con la palabra with
4.- Pueden combinar múltiples traits y redefinir si hay conflictos

Uso:

- Cuando quieras redefinir comportamientos comunes que puedan ser reutilizados en múltiples clases.
- Herencia múltiple de forma ordenada y controlada

Rasgos (Traits)

```
trait SangreFria

class Cocodrilo extends Animal with Carnivoro with SangreFria {
    ↑ TipoCriatura
  val TipoCriatura: String = "croc"
    ↑ comer
  def comer: Unit = println("ñam ñam")
    ↑ comer
  def comer(animal: Animal): Unit =
      println(s"Yo sou un cocodrilo y me como ${animal.TipoCriatura}")
}
val perro = new Perro
val croc = new Cocodrilo
croc.comer (perro)
```

Rasgos (Traits) vs Clases Abstractas

```
trait Carnivoro (name: String) {
def comer (animal: Animal): Unit
val prefiereComida: String ="Carne fresca"
}
```

¿Es correcta está implementación?

Rasgos (Traits) vs Clases Abstractas

```
// 1- rasgos (traits) no tienen parámetros constructores
   /* Ejemplo: no permite parámetros */
        trait Carnivoro (name: String) {
        def comer (animal: Animal): Unit
        val prefiereComida: String ="Carne fresca"
        }
```

ERROR TRAITS NO PUEDEN TENER PARÁMETROS

```
class Cocodrilo extends Animal with Carnivoro with SangreFria {
  ↑ TipoCriatura
  override val TipoCriatura: String = "croc"
  ↑ comer
  def comer: Unit = println("ñam ñam")
  ↑ comer
  def comer(animal: Animal): Unit =
      println(s"Yo sou un cocodrilo y me como ${animal.TipoCriatura}")
}
```

Rasgos (Traits) vs Clases Abstractas

	Clase abstracta	Trait
Declaración	- abstract class - puede tener constructores con parámetros	- trait - no puede tener parámetros constructores
Herencia múltiple	Solo puede heredar de una única clase abstracta (herencia simple)	Una clase puede mezclar múltiples traits usando extends y with. Herencia múltiple
Parámetros	Puede tener parámetros en el constructor y campos inicializados	No puede tener parámetros en el constructor Sí puede contener campos, pero deben inicializarse directamente o ser abstractos
Rendimiento	Más eficiente	Ligera sobrecarga
Uso	Jerarquía de clases, clases hijas compartan un comportamiento común. Es adecuada si necesitas constructores con parámetros	Comportamientos comunes o módulos reutilizables que pueden mezclases con múltiples clases. Herencia múltiple

Ejemplo

```
abstract class Animal(nombre: String) {
    def sonido(): String
}
class Perro(nombre: String) extends Animal(nombre) {
    def sonido():String = "Guau"
}

trait Volador {
    def volar(): String = "Estoy volando"
}
trait Nadador {
    def nadar(): String = "Estoy nadando"
}
class Pato extends Volador with Nadador
```

Ejercicio 1: Dispositivos electrónicos

```
/*Ejercicio 1. Dispositivos electrónicos
 El sistema de Dispositivos electrónicos debe realizar las siguientes acciones:
    - Encender: imprime por pantalla "El dispositivo está encendido"
    - Apagar: imprime por pantalla "El dispositivo está apagado"

  El dispositivo Teléfono debe realizar la acción de Llamar a un número
     imprimiendo por pantalla "Llamando al número [número]"
  El dispositivo Tablet deber realizar la acción de navegar Web
     imprimiendo por pantalla "Navengando a [url]"
*/
```

Ejercicio 2: Formas de pago

```
/*Ejercicio 2. Formas de pago
 Sistema de Formas de pago deber cumplir que:
    - Todas las formas de pago deben procesar el pago pasándole una cantidad.
    - La forma de pago Tarjeta de Crédito debe imprimir por pantalla:
      "Procesando pago de [cantidad] con tarjeta de crédito"
    - La forma de pago PayPal debe imprimir por pantalla:
      "Procesando pago de [cantidad] con PayPal"
    - La forma de pago Transferencia Bancaria debe imprimir por pantalla:
      "Procesnado pago de [cantidad] mediante transferencia bancaria"
*/
```

Ejercicio 3: Electrodomésticos Inteligentes

```
/*Ejercicio 3: Electrodomésticos Inteligentes
 El sistema de Electrodomésticos Inteligentes debe cumplir que:
    - Algunos electrodomésticos tienen la función de conexión WiFi que imprime por pantalla
      "Conectando el electrodomésticos a la red WiFi"
    - Algunos otros tienen la función de programación automática donde se les pasa el tiempo e
    imprime por pantalla "Programando electrodoméstico a las [tiempo]"

    El electrodoméstico Lavadora puede conectarse a WiFi y ser programada.
    El electrodoméstico Refrigerador puede conectarse a WiFi.
*/
```

Ejercicio 4: Robots Multifuncionales

```
/*Ejercicio 4: Robots Multifuncionales
Sistema para modelar diferentes tipos de robots que contienen las características de:
    - Modelo que devuelve el modelo del robot, por ejemplo, "ModeloX".
    - Bateria que devuelve el nivel de batería del robot, por ejemplo, 80.
    - Estado que devuelve el mensaje por pantalla de "El robot [modelo] tien [bateria]% de batería"

    Algunos de los modelos de robots pueden levantar peso imprimiendo por pantalla:
        "El robot está levantando [peso] kg."
    Algunos de los modelos de robots puede asistir a personas imprimiento por pantalla:
        "El robot está asistiendo a [persona]."



     Existe dos tipos de robot:
        - Robot que se denomina Obrero que realiza los trabajos pesados.
        - Robot que se denomina Asistente que realiza la asistencia a personas

*/
```