TECH
CONSULTING

Trait and Case Class Presentation

By: David Ishimwe Ruberamitwe

TECH CONSULTING

Introduction to scala trait

- A trait in Scala is a fundamental building block that allows you to define reusable code and is similar to a Java interface, but with additional capabilities.
- Traits can include both abstract and non-abstract methods, making them more flexible.
- They are primarily used to achieve multiple inheritance since Scala does not support multiple class inheritance due to the diamond problem.

TECH CONSULTING

Key Features of trait

- **Abstract and Non-Abstract Methods: Traits allow the declaration of abstract methods (without a body) as well as non-abstract methods (with implementation).**
- **Multiple Inheritance Support: Traits can be mixed in with classes and other traits to achieve multiple inheritance without the complications of extending multiple classes.**
- **Trait Instantiation: Unlike abstract classes, traits cannot be instantiated directly; they need to be mixed into classes or objects.**

TECH
CONSULTING

Sealed Traits in Scala

- A sealed trait restricts the places where it can be extended.
- All subclasses of a sealed trait must be defined in the same file as the trait itself.
- Sealed traits are used to enforce exhaustiveness checks, especially with pattern matching.
- Example use case: Sealed traits are often used in Algebraic Data Types (ADTs) where you define a closed set of possible values.

TECH
CONSULTING

Inheritance Rules

- Single Class, Multiple Traits: A class can extend one class but can implement multiple traits.
- Linearization: Scala resolves method calls by creating a linear order of traits, which avoids conflicts and ambiguities.
- Diamond Problem: Unlike Java, Scala avoids the diamond problem by ensuring a clear method resolution order (MRO).

TECH CONSULTING

Comparison of Trait with Java Interface

- Java Interface: Cannot have implemented methods (until Java 8 with default methods).
- Scala Trait: Can have both abstract and non-abstract methods.
- Java Interface Multiple Inheritance: Achieved with interfaces and default methods, but can get complicated.
- Scala Trait Multiple Inheritance: Simplifies multiple inheritance and avoids diamond problem via linearization.

TECH
CONSULTING

Benefits of Scala Traits

- Flexibility: Combines the flexibility of interfaces and classes.
- Multiple Inheritance: Provides a clean way to achieve multiple inheritance.
- Pattern Matching: Sealed traits are useful for safe pattern matching and exhaustive checking.

TECH
CONSULTING

Use Case 1 in real world

- Add logging behavior to various classes without duplicating code.
- Trait: Logger trait.

Example:

- A Logger trait can provide logging methods like info(), debug(), and error().
- Classes like DatabaseConnector, FileReader, and APIHandler can mix in this Logger trait to reuse logging logic.
- Benefit: Enables consistent logging across multiple classes without code repetition.

TECH
CONSULTING

Use Case 2 in real world

- Multiple Behaviors for Animals in a Zoo Simulation
- Use Case: Define shared behaviors among different animal classes.
- Trait: Swimmable and Flyable traits.

Example:

- A Swimmable trait can be used for animals like Dolphin and Fish.
- A Flyable trait can be applied to Bird and Bat classes.

TECH
CONSULTING

Use Case 3 in real world

- Reusable Database Connection Functionality
- Use Case: Reuse database connection logic across different classes.
- Example Traits:
- A DatabaseConnectable trait provides methods like connect() and disconnect() to a database.
- Classes like UserRepository, OrderRepository, and ProductRepository can mix in this trait to access the database.
- Benefit: Centralizes database connection logic, making it easier to

TECH CONSULTING

Use Case 4 in real world

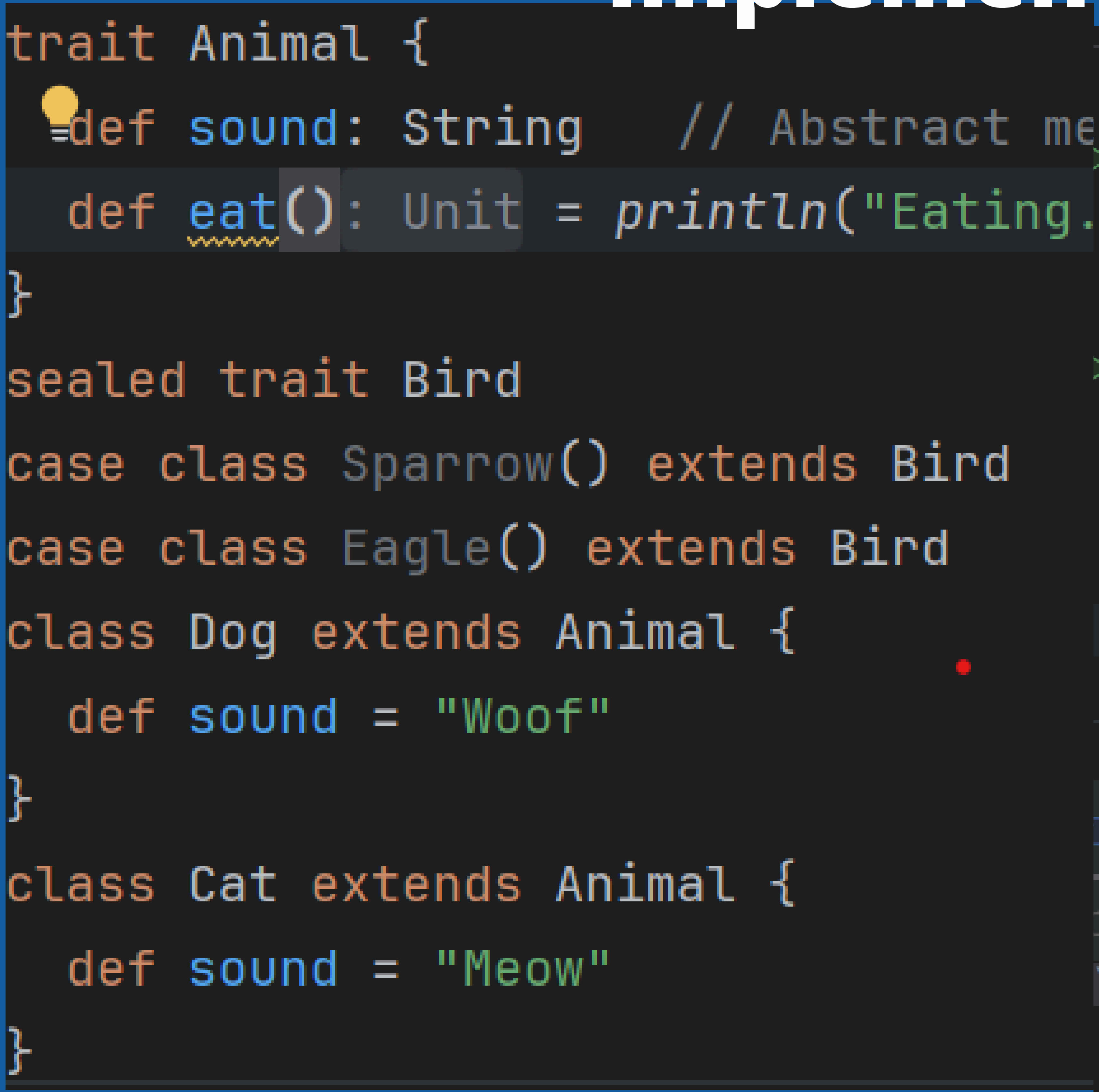
- Reusable Data Transformation Logic in Big Data Pipelines
- Use Case: A Big Data Developer needs to apply multiple transformation steps to different datasets in an Apache Spark project (e.g., cleaning, filtering, and aggregating data).

Example:

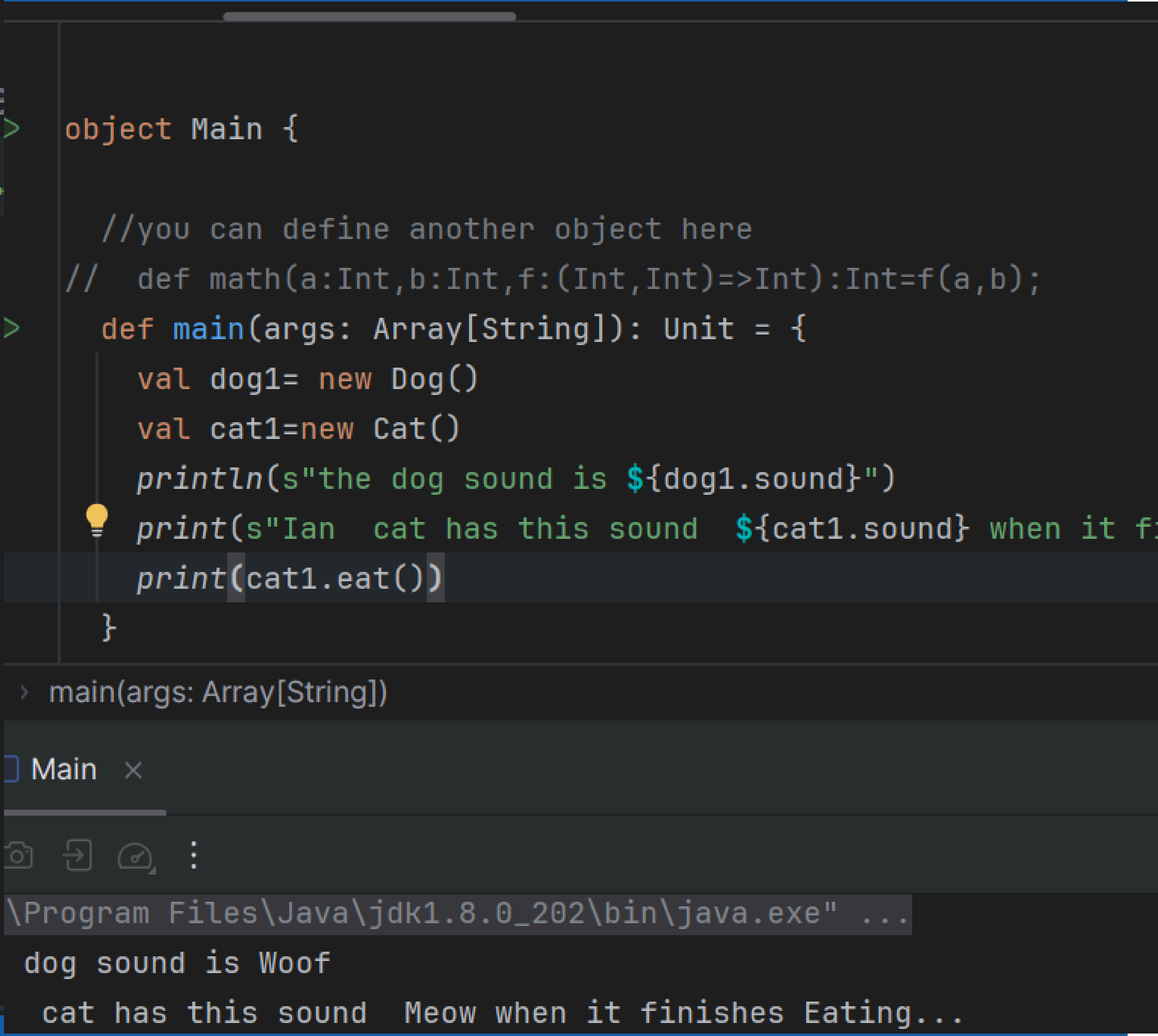
- trait DataCleaner: Provides methods for handling nulls, removing duplicates, etc.
- trait DataFilter: Offers methods for filtering based on specific conditions.
- trait Aggregator: Includes reusable aggregation functions (e.g., sum, average).
-

TECH CONSULTING

Implementation



```
trait Animal {
  def sound: String    // Abstract me
  def eat(): Unit = println("Eating.
}
sealed trait Bird
case class Sparrow() extends Bird
case class Eagle() extends Bird
class Dog extends Animal {
  def sound = "Woof"
}
class Cat extends Animal {
  def sound = "Meow"
}
```



```
object Main {

  //you can define another object here
//  def math(a:Int,b:Int,f:(Int,Int)=>Int):Int=f(a,b);
  def main(args: Array[String]): Unit = {
    val dog1= new Dog()
    val cat1=new Cat()
    println(s"the dog sound is ${dog1.sound}")
    print(s"Ian  cat has this sound  ${cat1.sound} when it f
    print(cat1.eat())
  }
```

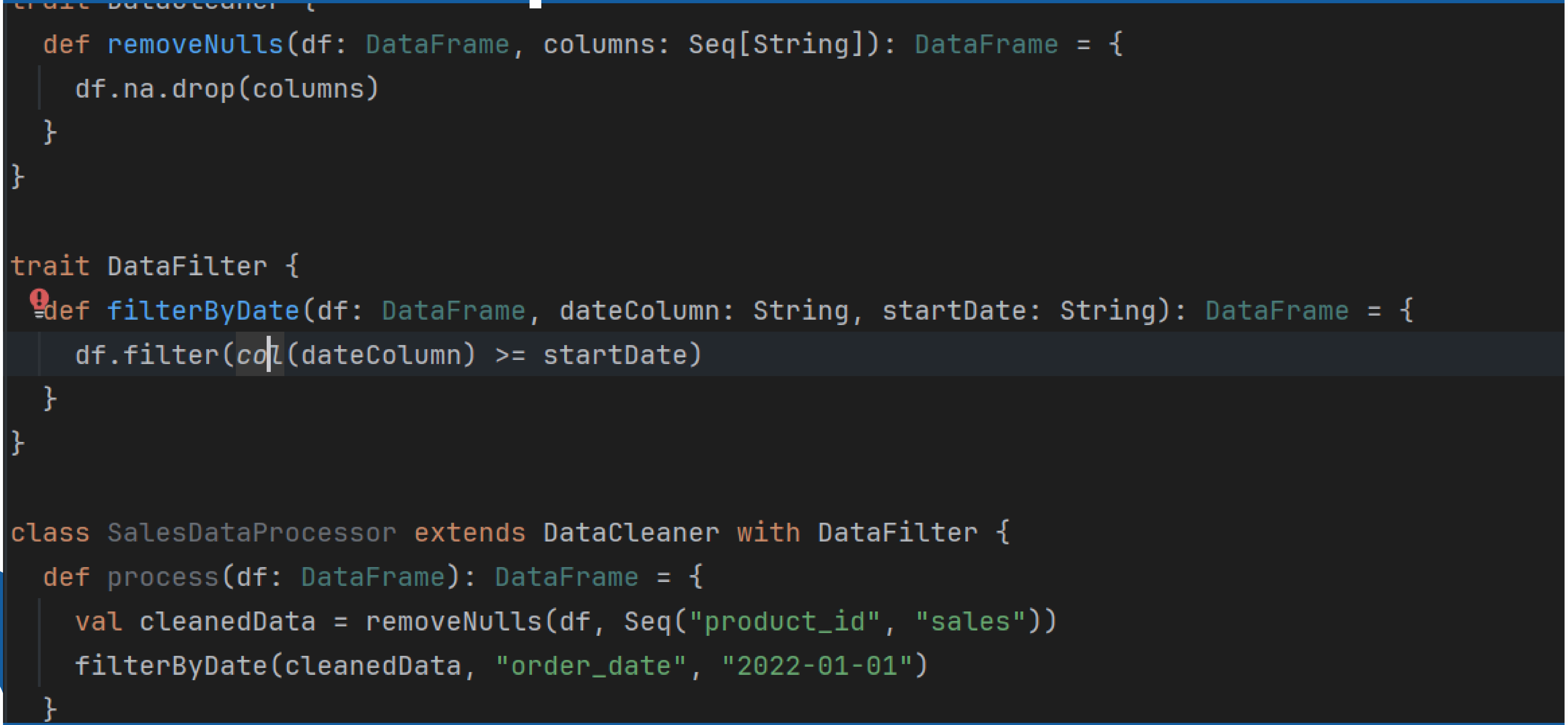
main(args: Array[String])

Main ×

```
\Program Files\Java\jdk1.8.0_202\bin\java.exe" ...
dog sound is Woof
 cat has this sound  Meow when it finishes Eating...
```

TECH
CONSULTING

Implementation 2



```
  def removeNulls(df: DataFrame, columns: Seq[String]): DataFrame = {
    df.na.drop(columns)
  }
}

trait DataFilter {
  def filterByDate(df: DataFrame, dateColumn: String, startDate: String): DataFrame = {
    df.filter(col(dateColumn) >= startDate)
  }
}

class SalesDataProcessor extends DataCleaner with DataFilter {
  def process(df: DataFrame): DataFrame = {
    val cleanedData = removeNulls(df, Seq("product_id", "sales"))
    filterByDate(cleanedData, "order_date", "2022-01-01")
  }
```

TECH
CONSULTING

Introduction to Case Classes

- Short Definition: A class that provides boilerplate code like equals, hashCode, and pattern matching automatically.
- Long Definition: In Scala, a case class is a special type of class that auto-generates methods such as equals, hashCode, toString, and supports pattern matching, making it ideal for functional programming. Andyou have to know that whenever a compiler see case key word that is when it generate code for you like these methods such as equals and more.

TECH CONSULTING

Characteristics

- Case classes are immutable by default.
- Provide concise syntax for data representation.
- no need of key word new to instantiate them.
- Case class constructor parameters are val fields by default, so an accessor method is generated for each parameter.
- You can never mutate its parameters.

TECH
CONSULTING

Benefits of case classes

- **Pattern Matching: Simplifies pattern matching, which is central to functional programming.**
- **Boilerplate Code Reduction means it automatically generates equals, hashCode, toString and copy methods.**
- **Immutable Data Representation: Case classes default constructor parameters to val, promoting immutability.**

TECH
CONSULTING

Use cases of case class

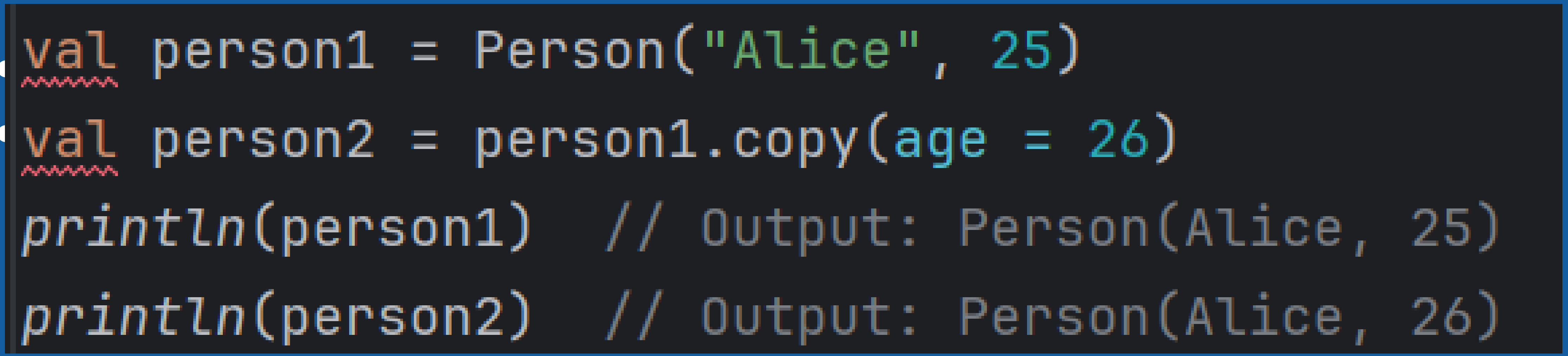
- **Data Modeling: Case classes are ideal for representing immutable data models, e.g., customer information, events in an event-driven system.**
- **Pattern Matching in Functional Programming: Useful in Algebraic Data Types (ADTs) and matching sealed traits to enforce exhaustive pattern matching.**
- **is used for schema matching**

TECH
CONSULTING

What most people don’t know

1.copy Method with Named Parameters

- What It Is: Case classes automatically generate a copy method that allows you to create a new instance with some fields changed while keeping the rest the same.
- Why It’s Useful: You can change only specific fields without manually reconstructing the entire object.
- The copy method can be very powerful when working with immutable data structures, as it allows you to create modified versions of data without changing the original.



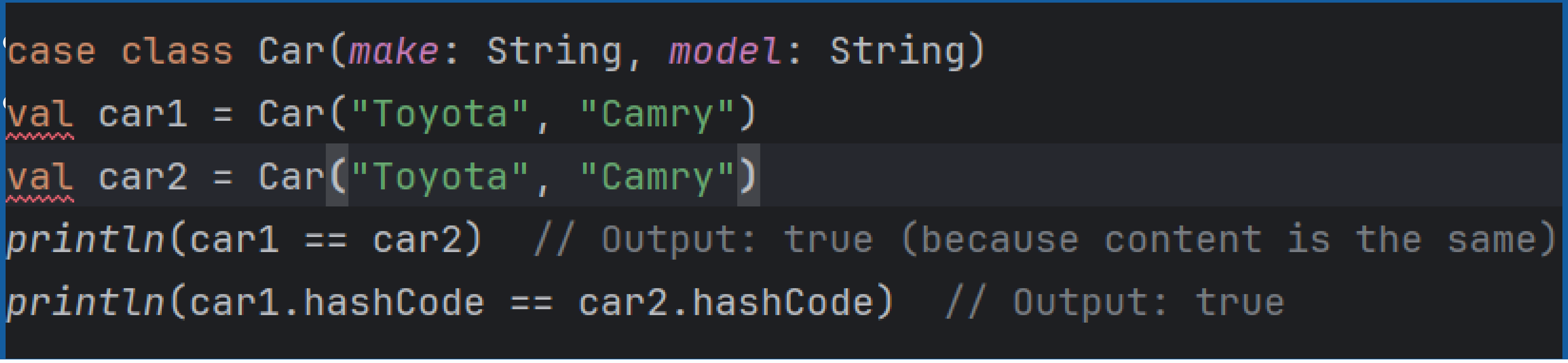
```
val person1 = Person("Alice", 25)
val person2 = person1.copy(age = 26)
println(person1)  // Output: Person(Alice, 25)
println(person2)  // Output: Person(Alice, 26)
```

TECH CONSULTING

What most people don’t know

2.Equality and hashCode Are Auto-Generated

- **What It Is: Case classes automatically generate equals and hashCode methods based on their parameters.**
- **Why It’s Useful: This makes comparisons and hash-based collections (like Set and Map) behave intuitively when using case classes.**
- **Unlike regular classes, you don’t need to override equals and hashCode for case classes. This can be particularly useful when using them as keys in a Map or storing them in a Set.**



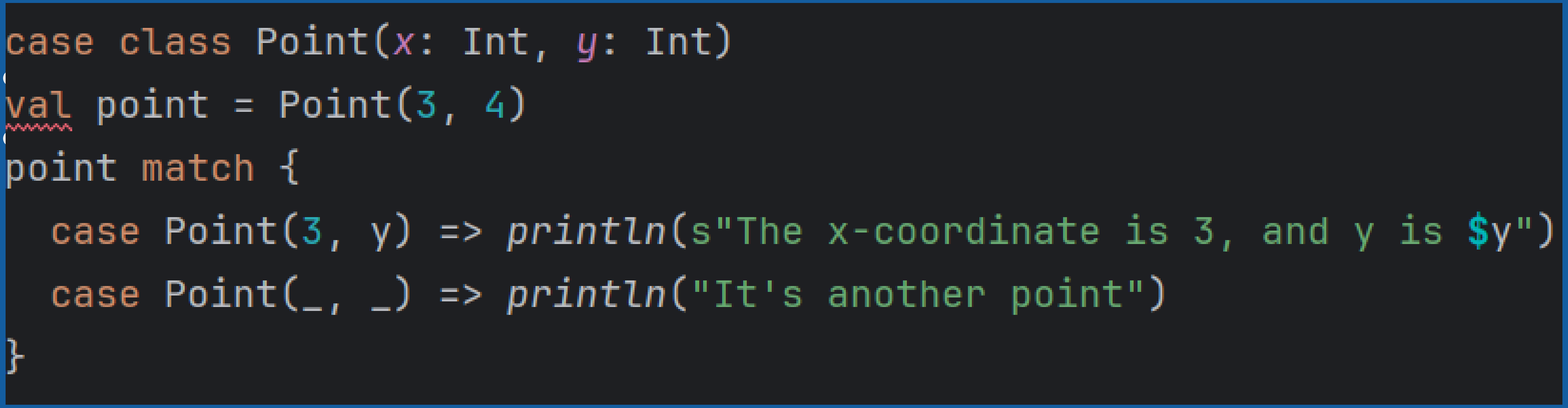
```scala
case class Car(make: String, model: String)
val car1 = Car("Toyota", "Camry")
val car2 = Car("Toyota", "Camry")
println(car1 == car2)  // Output: true (because content is the same)
println(car1.hashCode == car2.hashCode)  // Output: true
```

TECH CONSULTING

What most people don’t know

3.unapply Method for Pattern Matching

- **What It Is: Case classes automatically have a unapply method, making them ideal for pattern matching.**
- **Why It’s Useful: It allows you to decompose objects into their fields easily.**
- **The unapply method is what makes case classes work so well with match expressions. This feature is hidden behind the scenes, but it’s the reason why case classes are preferred for pattern matching.**



```scala
case class Point(x: Int, y: Int)
val point = Point(3, 4)
point match {
  case Point(3, y) => println(s"The x-coordinate is 3, and y is $y")
  case Point(_, _) => println("It's another point")
}
```

TECH
CONSULTING

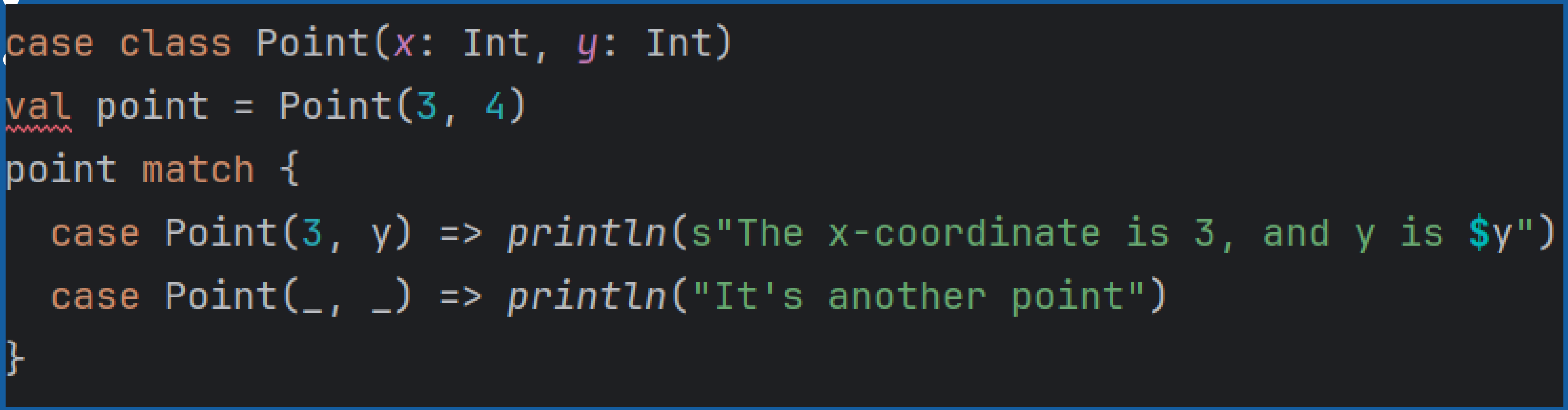
What most people don't know

4.Case Classes Have toString Automatically Implemented

What It Is: Case classes come with a pretty-printed toString method out of the box.

Why It's Useful: Makes debugging and logging easier because the output is more readable.

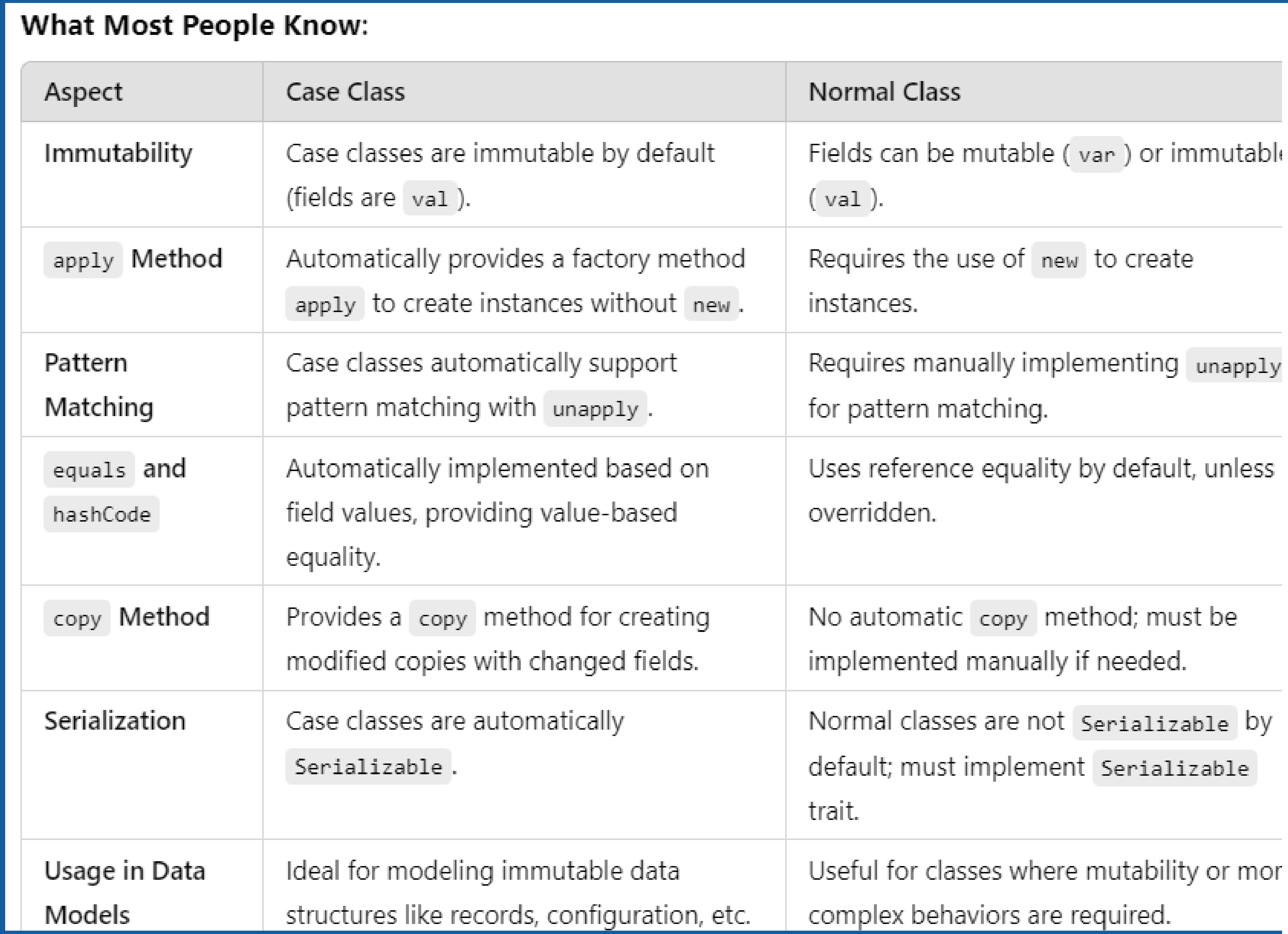
Unlike regular classes where you'd have to manually override toString, case classes do this automatically. This is very useful when working with collections of case class objects for quick inspection.



```scala
case class Point(x: Int, y: Int)
val point = Point(3, 4)
point match {
  case Point(3, y) => println(s"The x-coordinate is 3, and y is $y")
  case Point(_, _) => println("It's another point")
}
```

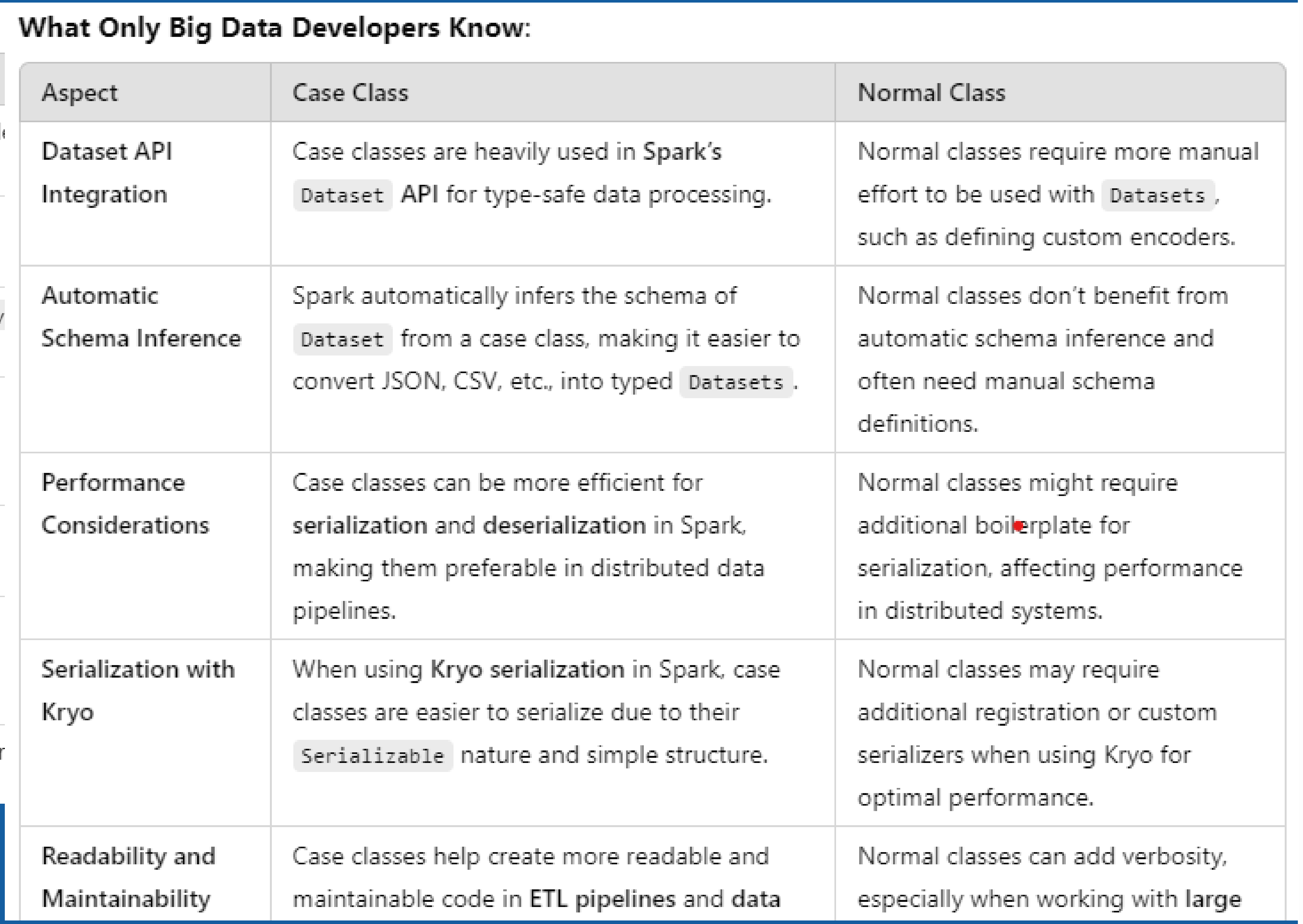
TECH CONSULTING

Differences Between Case Classes and Normal Classes in Scala



What Most People Know:

Aspect	Case Class	Normal Class
Immutability	Case classes are immutable by default (fields are `val`).	Fields can be mutable (`var`) or immutabl (`val`).
`apply` Method	Automatically provides a factory method `apply` to create instances without `new`.	Requires the use of `new` to create instances.
Pattern Matching	Case classes automatically support pattern matching with `unapply`.	Requires manually implementing `unapply` for pattern matching.
`equals` and `hashCode`	Automatically implemented based on field values, providing value-based equality.	Uses reference equality by default, unless overridden.
`copy` Method	Provides a `copy` method for creating modified copies with changed fields.	No automatic `copy` method; must be implemented manually if needed.
Serialization	Case classes are automatically `Serializable`.	Normal classes are not `Serializable` by default; must implement `Serializable` trait.
Usage in Data Models	Ideal for modeling immutable data structures like records, configuration, etc.	Useful for classes where mutability or mor complex behaviors are required.

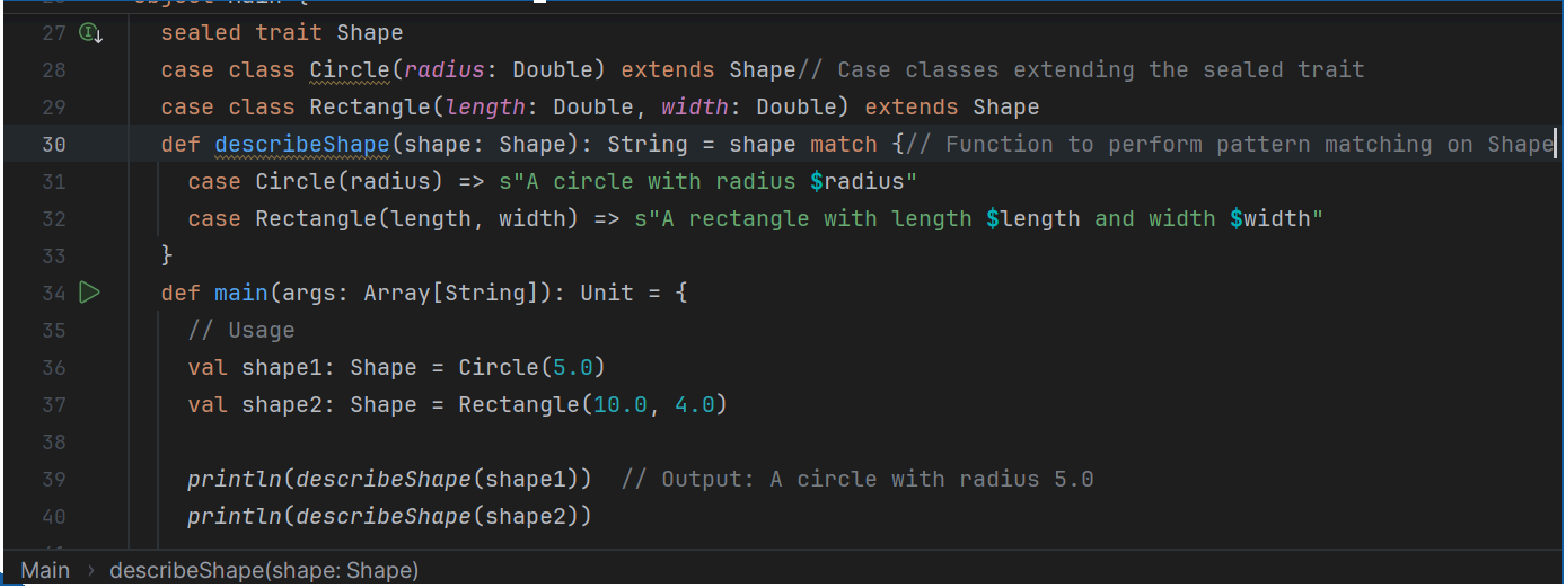


What Only Big Data Developers Know:

Aspect	Case Class	Normal Class
Dataset API Integration	Case classes are heavily used in Spark's `Dataset` API for type-safe data processing.	Normal classes require more manual effort to be used with `Datasets`, such as defining custom encoders.
Automatic Schema Inference	Spark automatically infers the schema of `Dataset` from a case class, making it easier to convert JSON, CSV, etc., into typed `Datasets`.	Normal classes don't benefit from automatic schema inference and often need manual schema definitions.
Performance Considerations	Case classes can be more efficient for **serialization** and **deserialization** in Spark, making them preferable in distributed data pipelines.	Normal classes might require additional boilerplate for serialization, affecting performance in distributed systems.
Serialization with Kryo	When using **Kryo serialization** in Spark, case classes are easier to serialize due to their `Serializable` nature and simple structure.	Normal classes may require additional registration or custom serializers when using Kryo for optimal performance.
Readability and Maintainability	Case classes help create more readable and maintainable code in **ETL pipelines** and **data**	Normal classes can add verbosity, especially when working with **large**

TECH
CONSULTING

Implementation



```scala
sealed trait Shape
case class Circle(radius: Double) extends Shape// Case classes extending the sealed trait
case class Rectangle(length: Double, width: Double) extends Shape
def describeShape(shape: Shape): String = shape match {// Function to perform pattern matching on Shape
  case Circle(radius) => s"A circle with radius $radius"
  case Rectangle(length, width) => s"A rectangle with length $length and width $width"
}
def main(args: Array[String]): Unit = {
  // Usage
  val shape1: Shape = Circle(5.0)
  val shape2: Shape = Rectangle(10.0, 4.0)

  println(describeShape(shape1))  // Output: A circle with radius 5.0
  println(describeShape(shape2))
```

Main › describeShape(shape: Shape)

TECH CONSULTING

Conclusion

- Traits in Scala allow the creation of reusable, modular behavior by enabling the definition of abstract and concrete methods that can be mixed into multiple classes.
- Case classes provide a concise way to model immutable data structures, automatically generating essential methods like equals, hashCode, copy, and toString.
- Traits support multiple inheritance, allowing for shared behavior across different classes without code duplication.
- Case classes are especially useful for pattern matching, making them a natural fit for functional programming in Scala.
- By using traits and case classes together, developers can write clean, maintainable, and scalable code.

TECH
CONSULTING

THANK YOU!