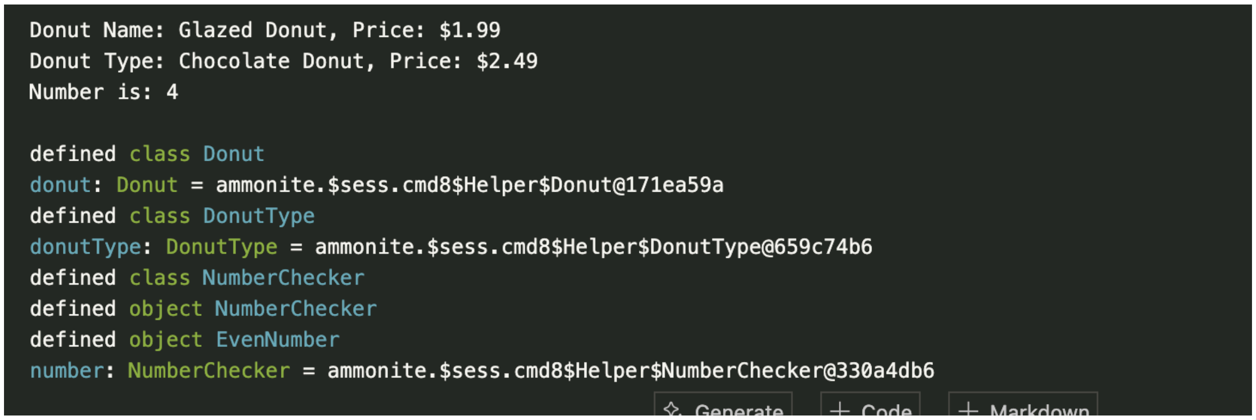
Day 2 - Concepts

1. Classes and Objects

```scala
// classes

class Donut(var name: String, val price: Double){
   // class body can contain methods and additional fields

   def display(): Unit = {
       println(s"Donut Name: $name, Price: $$${price}")
   }
}

val donut = Donut("Glazed Donut", 1.99)
donut.display() // calling method with parentheses - standard way
donut.name = "Choco Donut"


class DonutType(val name: String, val price: Double){

   def display: Unit = {
       println(s"Donut Type: $name, Price: $$${price}")
   }
}

val donutType = new DonutType("Chocolate Donut", 2.49)
donutType.display // calling method without parentheses - valid in Scala only if
method has no parameters or () is omitted

// pattern matching with unapply method

class NumberChecker(val number: Int) {
 def isEven: Boolean = number % 2 == 0
}

object NumberChecker {
 def unapply(n: NumberChecker): Option[Int] = Some(n.number)
}
```

```scala
object EvenNumber {
 def unapply(x: NumberChecker): Option[Int] =
   if (x.isEven) Some(x.number) else None // using unapply method to check
NumberChecker object.
}

val number = new NumberChecker(4)

number match { // pattern matching that matches objects based on unapply methods
 case NumberChecker(n) => println(s"Number is: $n") //executes the first matching case
 case EvenNumber(n) => println(s"$n is even")
 case _               => println(s"${number.number} is odd")
}
```

Output:



```
Donut Name: Glazed Donut, Price: $1.99
Donut Type: Chocolate Donut, Price: $2.49
Number is: 4

defined class Donut
donut: Donut = ammonite.$sess.cmd8$Helper$Donut@171ea59a
defined class DonutType
donutType: DonutType = ammonite.$sess.cmd8$Helper$DonutType@659c74b6
defined class NumberChecker
defined object NumberChecker
defined object EvenNumber
number: NumberChecker = ammonite.$sess.cmd8$Helper$NumberChecker@330a4db6
```

2. Case Classes

```scala
// case classes

// case class will automatically create companion objects.
// instance variables are immutable in nature.

case class Person(name: String, age: Int, code: Option[String] = None)

val alice = Person("Alice", 30, Some("A123"))
println(alice.name) // Output: Alice
println(alice.age)  // Output: 30
println(alice.code) // Output: Some(A123)
```

```scala
val bob = alice.copy(name = "Bob")
println(bob.name) // Output: Bob
println(bob.age)  // Output: 30
println(bob.code) // Output: Some(A123)

// pattern matching with case classes

def greetPerson(person: Person): String = person match {
 case Person(name, age, code) if age < 18 => s"Hello, young $name!"
 case Person(name, age, code) => s"Hello, $name!"
}

println(greetPerson(alice)) // Output: Hello, Alice!
println(greetPerson(bob))   // Output: Hello, Bob!
println(greetPerson(Person("Charlie", 25))) // Output: Hello, Charlie!
println(greetPerson(Person("Daisy", 15)))   // Output: Hello, young Daisy!

// object equals in case classes

val a = Person("a",10)
val b = Person("a", 10) // equality works based on value not address.

println(a == b)
```

Output:

```
Alice
30
Some(A123)
Bob
30
Some(A123)
Hello, Alice!
Hello, Bob!
Hello, Charlie!
Hello, young Daisy!
true

defined class Person
alice: Person = Person(name = "Alice", age = 30, code = Some(value = "A123"))
bob: Person = Person(name = "Bob", age = 30, code = Some(value = "A123"))
defined function greetPerson
a: Person = Person(name = "a", age = 10, code = None)
b: Person = Person(name = "a", age = 10, code = None)
```

3. Apply and Unapply method, Companion Object

```
class Person(val name: String, val age: Int)

object Person {
 def apply(name: String, age: Int): Person = new Person(name, age)
 def apply(name: String): Person = new Person(name, 0)
 def apply(): Person = new Person("Unknown", 0)
 def unapply(p: Person): Option[(String, Int)] = Some((p.name, p.age))
}

val p1 = Person("Alice", 30)   // uses apply(name: String, age: Int)
val p2 = Person("Bob")          // uses apply(name: String)
val p3 = Person()               // uses apply()
println(s"${p1.name}, ${p1.age}") // Alice, 30
println(s"${p2.name}, ${p2.age}") // Bob, 0
println(s"${p3.name}, ${p3.age}") // Unknown, 0

val person = Person("Charlie", 25)

person match
 case Person(name, age) => println(s"Name: $name, Age: $age")
```

Output:

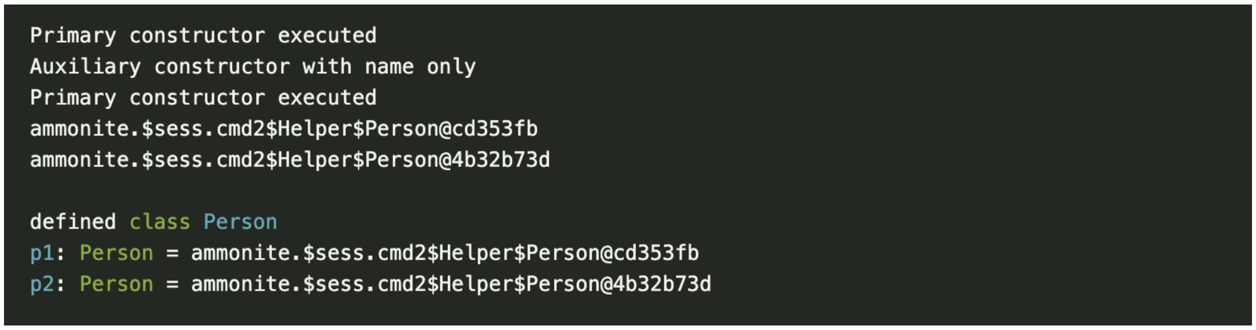
```
Alice, 30
Bob, 0
Unknown, 0
Name: Charlie, Age: 25

defined class Person
defined object Person
p1: Person = ammonite.$sess.cmd1$Helper$Person@53fc0d83
p2: Person = ammonite.$sess.cmd1$Helper$Person@61a691e0
p3: Person = ammonite.$sess.cmd1$Helper$Person@421d2957
person: Person = ammonite.$sess.cmd1$Helper$Person@4030dd2a
```

4. Auxillary Constructor

```
// auxillary constructors

class Person(val name: String, val age: Int) {
```

```scala
 println("Primary constructor executed")

 // Auxiliary constructor 1
 def this(name: String) = {
   this(name, 0) // calling primary constructor
   println("Auxiliary constructor with name only")
 }

 // Auxiliary constructor 2
 def this() = {
   this("Unknown", 0) // calling another auxiliary constructor
   println("Auxiliary constructor with no args")
 }
}

val p1 = new Person("Scala") // new is needed because of auxiliary constructor
val p2 = new Person("Programming", 12)
println(p1)
println(p2)
```

Output:



```
Primary constructor executed
Auxiliary constructor with name only
Primary constructor executed
ammonite.$sess.cmd2$Helper$Person@cd353fb
ammonite.$sess.cmd2$Helper$Person@4b32b73d

defined class Person
p1: Person = ammonite.$sess.cmd2$Helper$Person@cd353fb
p2: Person = ammonite.$sess.cmd2$Helper$Person@4b32b73d
```

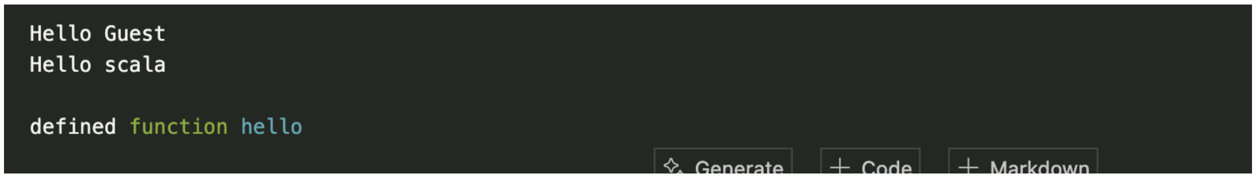
5. Operator Overloading

```scala
// operator overloading

class StringEnhancer(val str: String) {
   // Overload unary '!' operator to append a character, e.g., '!'
   def unary_! : String = str + "!"
}

object OperatorOverloadingDemo extends App {
   val enhanced = new StringEnhancer("Hello")
   println(!enhanced) // Output: Hello!
}
```

6. Methods with default Parameters

```
def hello(input: String = "Guest"): Unit = {
   println(s"Hello $input")
}

hello()
hello("scala")
```

Output:



```
Hello Guest
Hello scala

defined function hello
```

7. Method Overloading

```
// methods
def greet(name: String): String = s"Hello, $name!"

// method overloading
def add(x: Int, y: Int): Int = x + y
def add(x: Double, y: Double): Double = x + y
def add(x: String, y: String): String = x + y

println(greet("World")) // Output: Hello, World!
println(add(5, 10)) // Output: 15
println(add(5.5, 10.2)) // Output: 15.7
println(add("Hello, ", "Scala!")) // Output: Hello, Scala!
```

Output:

```
Hello, World!
15
15.7
Hello, Scala!

defined function greet
defined function add
defined function add
defined function add
```