Object-oriented programming with Scala

Object-oriented programming (OOP)

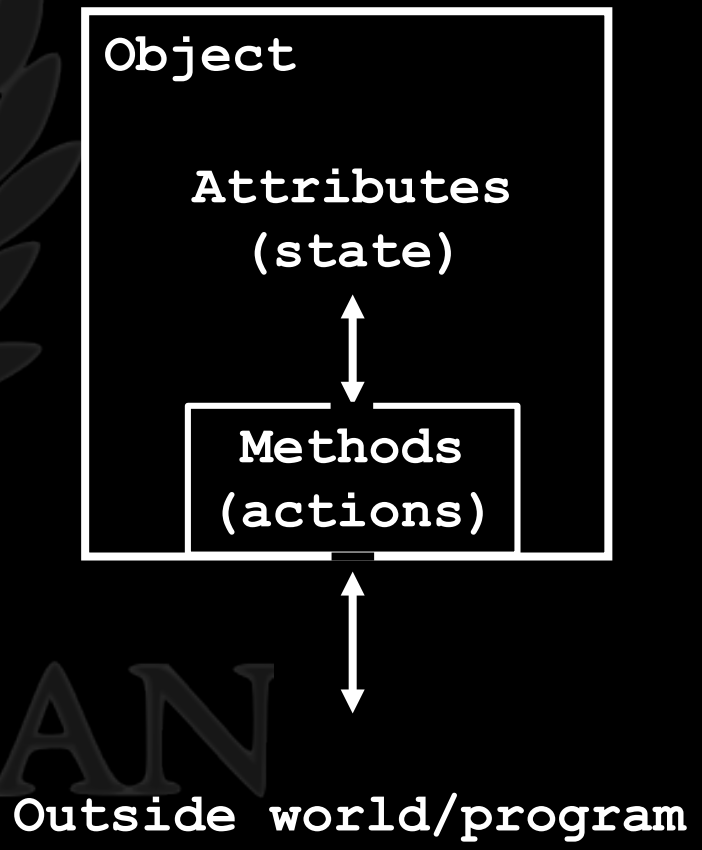
It's a paradigm which tries to model objects/things in terms of <u>attributes</u> and <u>actions</u>.

To model objects/things we use <u>classes</u>.

Object-oriented programming (OOP)

Objects/things encapsulate attributes and actions e.g.:

- A dog has age and breed (attributes).
- A dog can run and bark (actions).



Classes

To model <u>objects/things</u> we use classes.

They can be thought as templates from which objects are created e.g.:

- Snoopy and Lassie are dogs (they have dog attributes and actions).
- A tweet has a text (attribute) and can be retweeted (action).

Classes

```
class Person

val jenny = new Person
```

Instantiates
an object 'a'

Classes

```
class Person {
  var weight: Double = 58
```

→ **Attribute.** State of the object

```
  def exercise(min: Double) {
    weight -= min * .0014
  }
}
```

→ **Method.** Access to the state

Classes

```scala
class Person {
  var weight: Double = 58


  def exercise(min: Double) {
    weight -= min * .0014
  }
}
```

```scala
val jenny = new Person
jenny.weight      // = 58.0
jenny.exercise(30)
jenny.weight      // = 57.958
```

Classes: constructors

```
class Person (w: Double) {
  var weight: Double = w


  def exercise(min: Double) {
    weight -= min * .0014
  }
}
```

```
val jenny = new Person(58)
val jack = new Person(74)
jenny.weight      // = 58.0
jack.weight       // = 74.0
```

Classes: constructors

```
class Person (var weight: Double) {

  def exercise(min: Double) {
    weight -= min * .0014
  }
}
```

```
val jenny = new Person(58)
val jack = new Person(74)
jenny.weight      // = 58.0
jack.weight       // = 74.0
```

Classes: constructors (auxiliary)

```
class Person (var weight: Double) {
  def this(other: Person) {
    this(other.weight)
  }
  def exercise(min: Double) {
    weight -= min * .0014
  }
}
```

```
val jenny = new Person(58)
val jack = new Person(jenny)
jenny.weight      // = 58.0
jack.weight       // = 58.0
```

Classes: val vs. var

```scala
class Person (var weight: Double)
```

```scala
val jenny = new Person(58)
val jack = new Person(72)
jenny.weight = 60.0      // ??
jack.weight = 75.0       // ??
```

Classes: val vs. var

```scala
class Person (var weight: Double)
```

```scala
val jenny = new Person(58)
val jack = new Person(72)
jenny = jack        // ??
```

OOP principle

- Attributes define a state.
- The state can only be seen or modified through methods.

Classes: private modifier

```
class Person (private var name: String)
```

```
val jenny = new Person("Jenny")
jenny.name  // = ERROR
```

Classes: private modifier

```
class Person (private var name: String) {
  def greet() {
    println("Hi, I'm " + name)
  }
}
```

```
val jenny = new Person("Jenny")
jenny.greet()  // = Hi, I'm Jenny
```

Classes: private modifier

```scala
class Person (private var name: String) {
  def greet() {
    println("Hi, I'm " + name)
  }
  def setName(n: String) {
    name = n
  }
}
```

```scala
val jenny = new Person("Jenny")
jenny.greet()  // = Hi, I'm Jenny
jenny.setName("Yenni")
jenny.greet()  // = Hi, I'm Yenni
```

Objects: logic

```
class Person (var name: String)
```

```
val jenny = new Person("Jenny")
```

Objects: logic

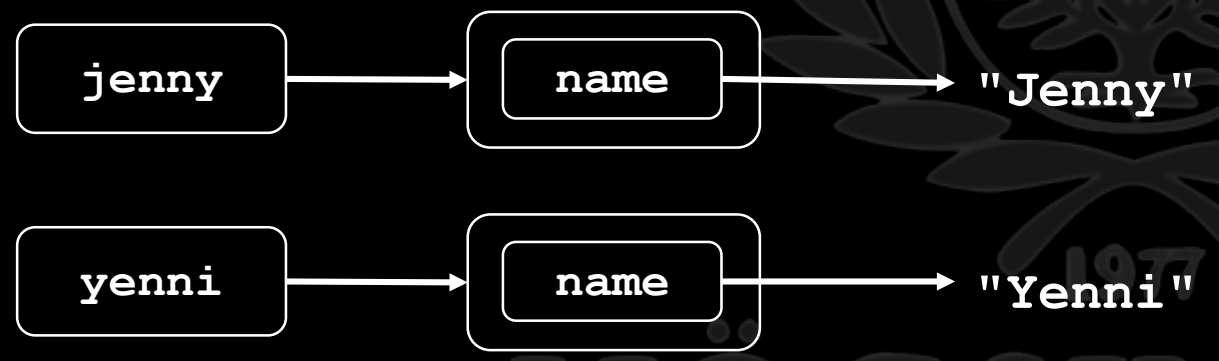
```
class Person (var name: String)
```

```
val jenny = new Person("Jenny")
jenny.name = "Yenni"
```

Objects: logic

```
class Person (val name: String)
```

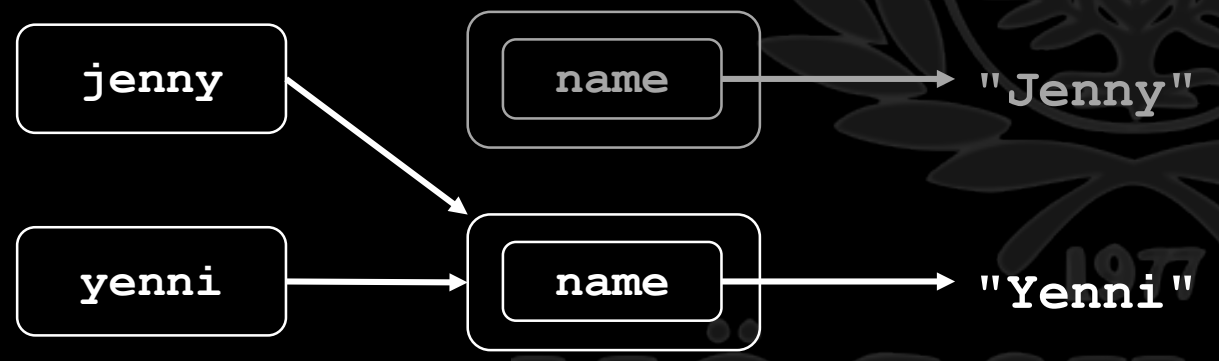
```
var jenny = new Person("Jenny")
var yenni = new Person("Yenni")
```



Objects: logic

```
class Person (val name: String)
```

```
var jenny = new Person("Jenny")
var yenni = new Person("Yenni")
jenny = yenni
```



Singleton objects

- A singleton object is a simplified way of implementing the singleton pattern.
- Used when there's no need of more than one instance.
- Used to wrap static fields and methods.

Singleton objects

```
object Calc {
  def abs(x: Double): Double =
    if (x < 0) -x
    else x
}
```

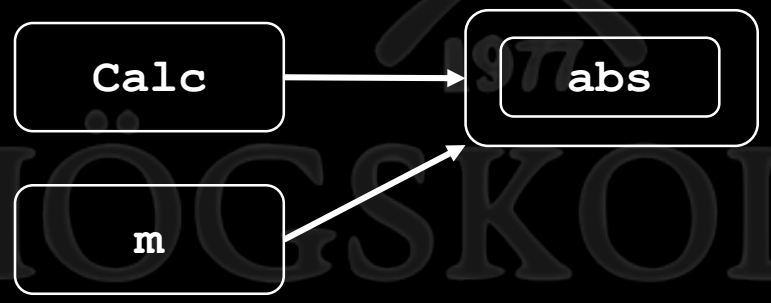
```
Calc.abs(-5)  // = 5.0
```

No parameters, no instantiation

Singleton objects

```
object Calc {
  def abs(x: Double): Double =
    if (x < 0) -x
    else x
}
```

```
val m = Calc
```



Singleton objects: as Main

```scala
object App {
  def main(args: Array[String]) {
    println("This is a Scala App")
  }
}
```

```
> scalac App.scala
> scala App
This is a Scala Application
```

Singleton objects: as Companions

```
object Person {
  var lastId = 0
  def nextId(): Int = {
    lastId += 1
    lastId
  }
}
```

```
class Person (val name: String) {
  val id: Int = Person.nextId()
  def greet() {
    println("Hi, I'm " + name + " ID: " + id)
  }
}
```

Singleton objects: as Companions

```
private object Person {
  var lastId = 0
  def nextId(): Int = {
    lastId += 1
    lastId
  }
}
```

```
class Person (val name: String) {
  val id: Int = Person.nextId()
  def greet() {
    println("Hi, I'm " + name + " ID: " + id)
  }
}
```

Singleton objects: as Companions

```
class Person private (val name: String)
object Person {
  var people: List[Person] = Nil
  def create(n: String): Person = {
    people = new Person(n) :: people
    people.head
  }
  def exists(n: String): Boolean = people.exists(_.name == n)
}
```

Composition

```
class Car (val brand: String)
class House (val address: String, val postnr: Int)
class Person (val name: String) {
  val car = new Car("Volvo")
  val home = new House("Hogskolevagen", 54128)
}
val jenny = new Person("Jenny")
jenny.car.brand    // = "Volvo"
jenny.home.address // = "Hogskolevagen"
```

Composition

```
class Course (val name: String)
class Student (val name: String) {
  var courses: List[Course] = Nil
  def enroll(c: Course) {
    courses = c :: courses
  }
}
```

Inheritance

```
class Person
class Employee extends Person


val jenny = new Employee
val jack = new Person
```

Inheritance

```
class Person (val name: String)
class Employee (n: String) extends Person(n)
```

```
val jenny = new Employee("Jenny")
val jack = new Person("Jack")
jenny.name  // = Jenny
jack.name   // = Jack
```

Inheritance

```
class Person (val name: String)
class Employee (val salary: Double, n: String) extends Person(n)
```

```
val jenny = new Employee(32000, "Jenny")
```

```
jenny.name        // = Jenny
jenny.salary      // = 32000.0
```

Inheritance: protected modifier

```
class Person (private val name: String)
class Employee (val salary: Double, n: String) extends Person(n)



val jenny = new Employee(32000, "Jenny")
jenny.name  // = ERROR
```

Inheritance: protected modifier

```
class Person (protected val name: String)
class Employee (val salary: Double, n: String) extends Person(n)
```

```
val jenny = new Employee(32000, "Jenny")
jenny.name  // = Jenny
```

Inheritance: abstract classes

```
abstract class Shape (val name: String) {
  def area: Double
  def describe: String = s"A $name with an area of $area"
}
class Square (val side: Double) extends Shape("Square") {
  def area: Double = side * side
}
val a = new Square(2)
a.area       // 4.0
a.describe   // A Square with an area of 4.0
```

Inheritance: abstract classes

```
abstract class Shape (val name: String) {
  def area: Double
  def describe: String = s"A $name with an area of $area"
}
class Square (val side: Double) extends Shape("Square") {
  def area: Double = side * side
}
val a = new Shape("Square")   // = ERROR
```

Inheritance: abstract classes

```
abstract class Shape (val name: String) {
  def area: Double
  def describe: String = s"A $name with an area of $area"
}

val a = new Shape("shape") {
  def area: Double = 4
}
```

Inheritance: traits

```
trait Describable {
  def describe: String
}
class Square (var side: Double) extends Describable {
  def describe: String = s"Square with sides = $side"
}

val a = new Square(2)
a.describe  // = Square with sides = 2.0
```

Inheritance: traits

```
trait Describable { def describe: String }
trait Resizable { def resize(area: Double) }

class Square (var side: Double) extends Describable with Resizable {
  def describe: String = s"Square with sides = $side"
  def resize(area: Double){ side = math.sqrt(area) }
}
val a = new Square(2)
a.resize(9)
a.describe  // = Square with sides = 3.0
```

Inheritance: traits

```
trait Describable { def describe: String }
trait Resizable { def resize(area: Double) }

class Square (var side: Double)

val a = new Square(2) with Describable {
  def describe: String = s"Square with sides = $side"
}
a.describe  // = Square with sides = 2.0
```

Inheritance: traits

```
trait Describable { def describe: String }
trait Resizable { def resize(area: Double) }

val a = new Describable {
  def describe: String = "A describable"
}

a.describe  // = A describable
```

Inheritance: traits

```
trait Describable { def describe: String }
trait Resizable { def resize(area: Double) }

val a = new Describable with Resizable {
  def describe: String = "A describable"
  def resize(area: Double){ throw new Exception("Nothing to resize") }
}
a.describe  // = A describable
a.resize(4) // = java.lang.Exception: Nothing to resize
```

Subtyping polymorphism

```
class Person (val name: String)
class Employee (val salary: Double, n: String) extends Person(n)
```

```
val jenny: Person = new Employee(32000, "Jenny")
jenny.name          // = ???
jenny.salary        // = ???
```

Subtyping polymorphism

```
class Person (val name: String)
class Employee (val salary: Double, n: String) extends Person(n)



val jenny: Person = new Employee(32000, "Jenny")
val salary = jenny match {
  case e: Employee => e.salary
}
```

Subtyping polymorphism

```
trait Describable { def describe: String }
trait Resizable { def resize(area: Double) }

class Square (var side: Double) extends Describable with Resizable {
  def describe: String = s"Square with sides = $side"
  def resize(area: Double){ side = math.sqrt(area) }
}
val a: Describable = new Square(2)
a.side      // = ???
a.describe  // = ???
```

Scala class hierarchy