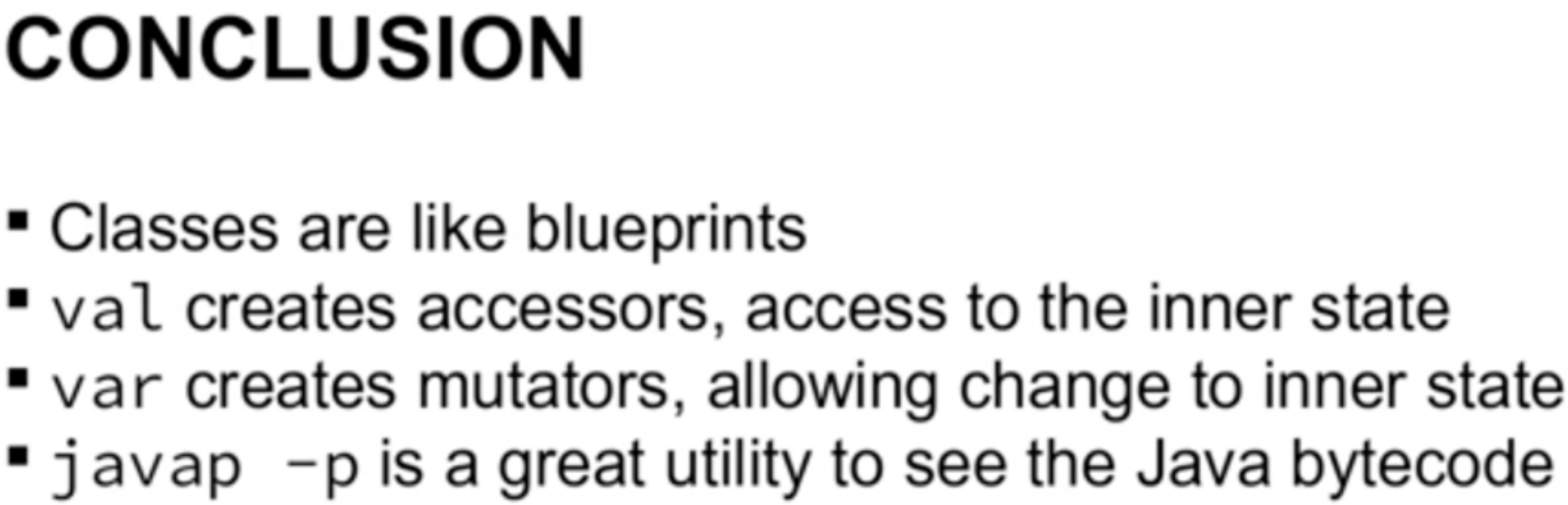
Module 2 begins

Classes in Scala



CONCLUSION

- Classes are like blueprints
- `val` creates accessors, access to the inner state
- `var` creates mutators, allowing change to inner state
- `javap -p` is a great utility to see the Java bytecode

Java Getters and Setters

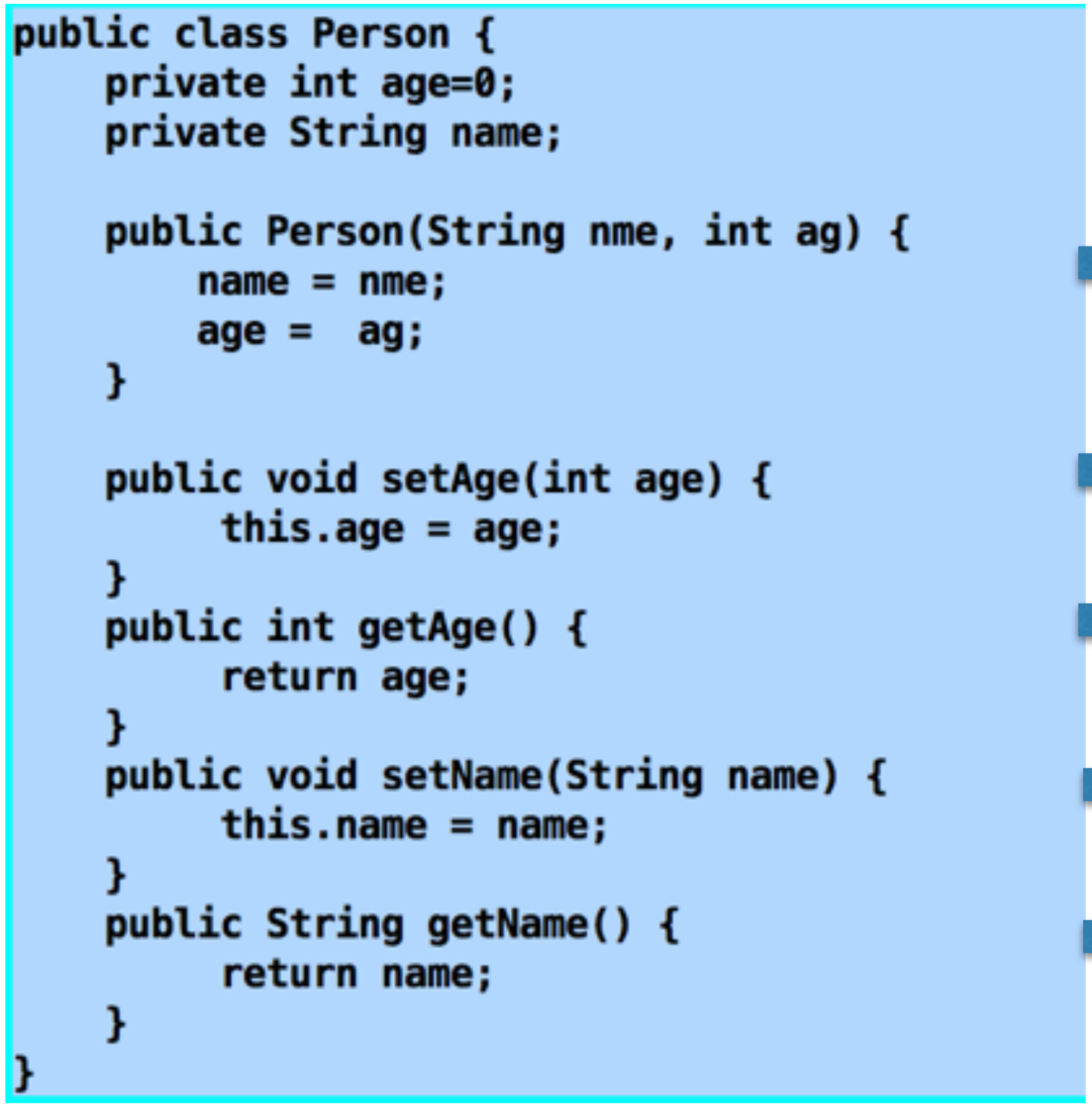
in JAVA person class should have below structure:

```
public class Person {
        private int age;
            public void setAge(int age) {
                this.age = age;
            }
            public int getAge() {
                return age;
            }
}
```

example in: ~/spark-1.6.1/scalatest/Person.java

- Step1: Javac Person.java
- Step2: javap -p Person (compare it with java -p Employee scala class which is having val and var attributes)
- Step3: java Person (to run and see the result)
- use of annotation look at Employee.scala

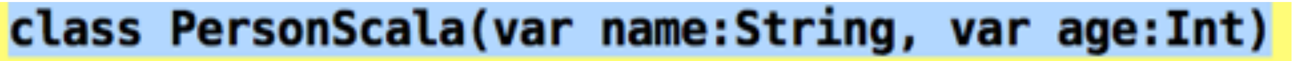
JAVA Class Declaration vs Scala Class Declaration



```
public class Person {
    private int age=0;
    private String name;

    public Person(String nme, int ag) {
        name = nme;
        age =  ag;
    }

    public void setAge(int age) {
         this.age = age;
    }
    public int getAge() {
         return age;
    }
    public void setName(String name) {
         this.name = name;
    }
    public String getName() {
         return name;
    }
}
```



```
class PersonScala(var name:String, var age:Int)
```

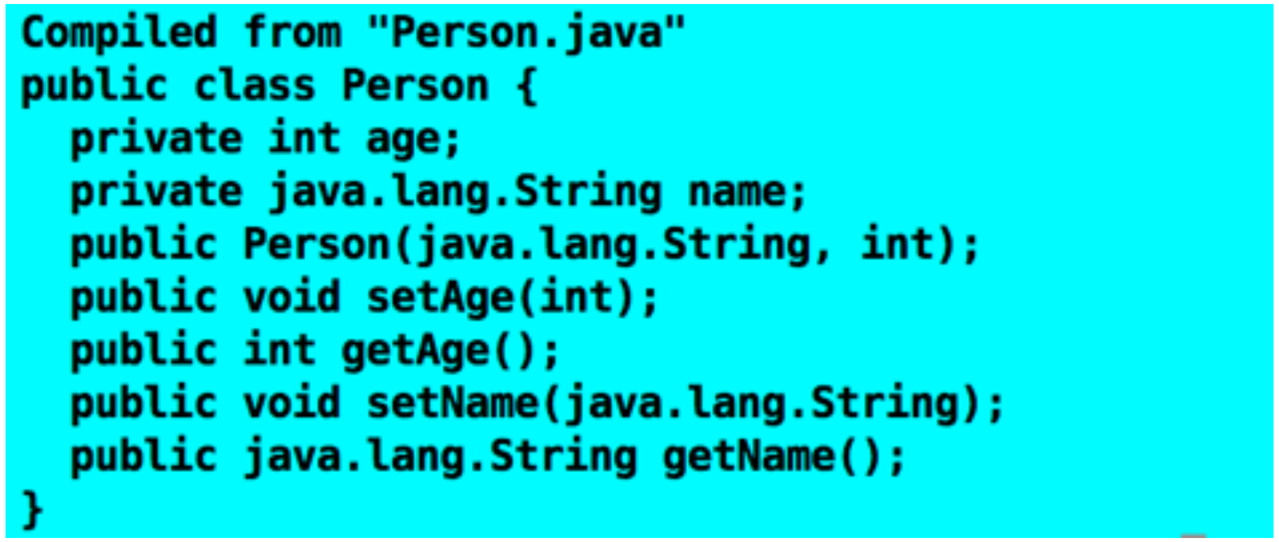
Constructor

Setter Function

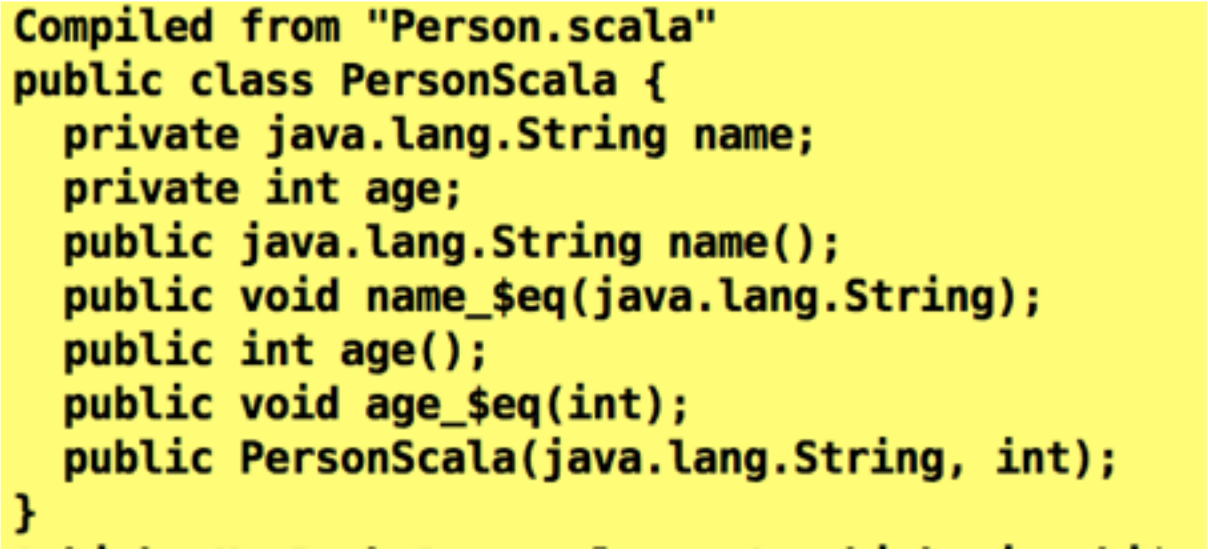
Getter Function

Setter Function

Getter Function

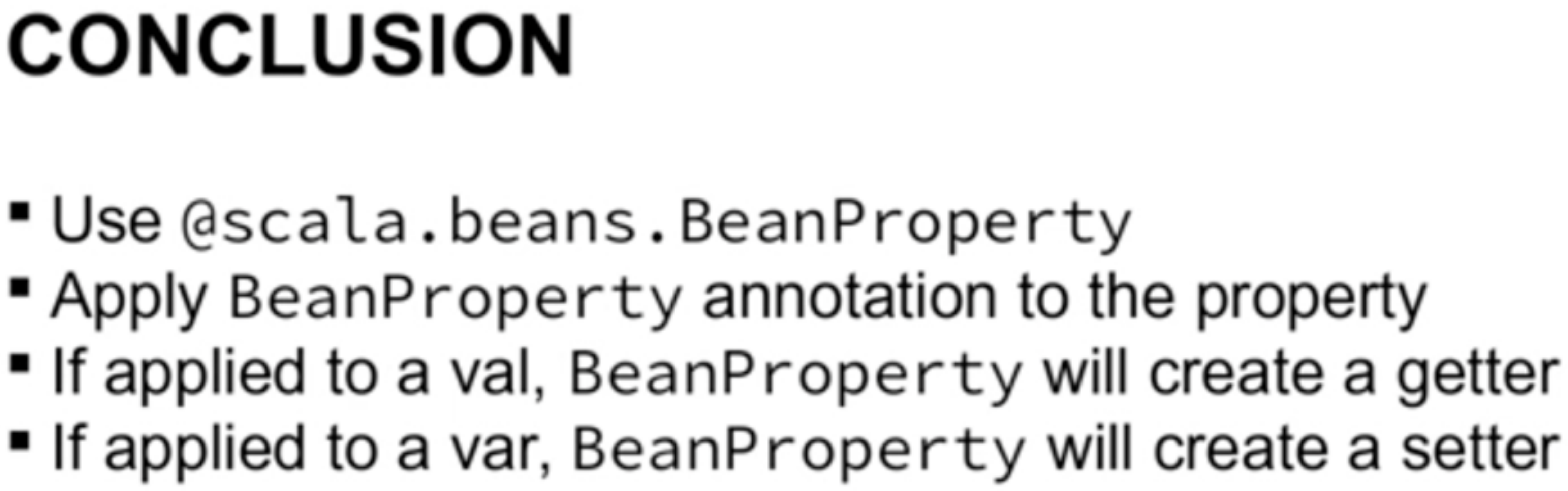


```
Compiled from "Person.java"
public class Person {
  private int age;
  private java.lang.String name;
  public Person(java.lang.String, int);
  public void setAge(int);
  public int getAge();
  public void setName(java.lang.String);
  public java.lang.String getName();
}
```



```
Compiled from "Person.scala"
public class PersonScala {
  private java.lang.String name;
  private int age;
  public java.lang.String name();
  public void name_$eq(java.lang.String);
  public int age();
  public void age_$eq(int);
  public PersonScala(java.lang.String, int);
}
```

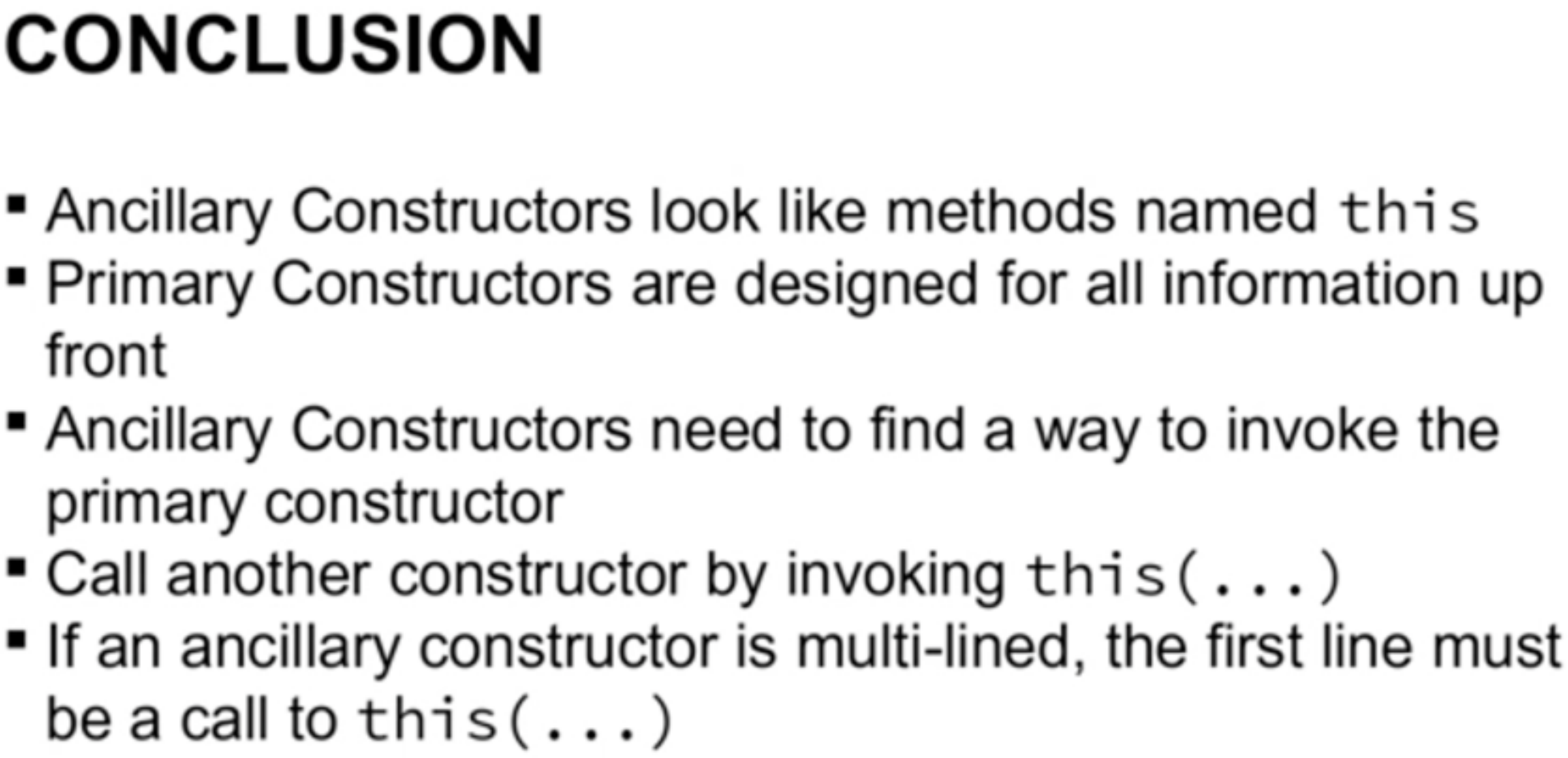
Scala Getters\Setters



CONCLUSION

- Use `@scala.beans.BeanProperty`
- Apply `BeanProperty` annotation to the property
- If applied to a val, `BeanProperty` will create a getter
- If applied to a var, `BeanProperty` will create a setter

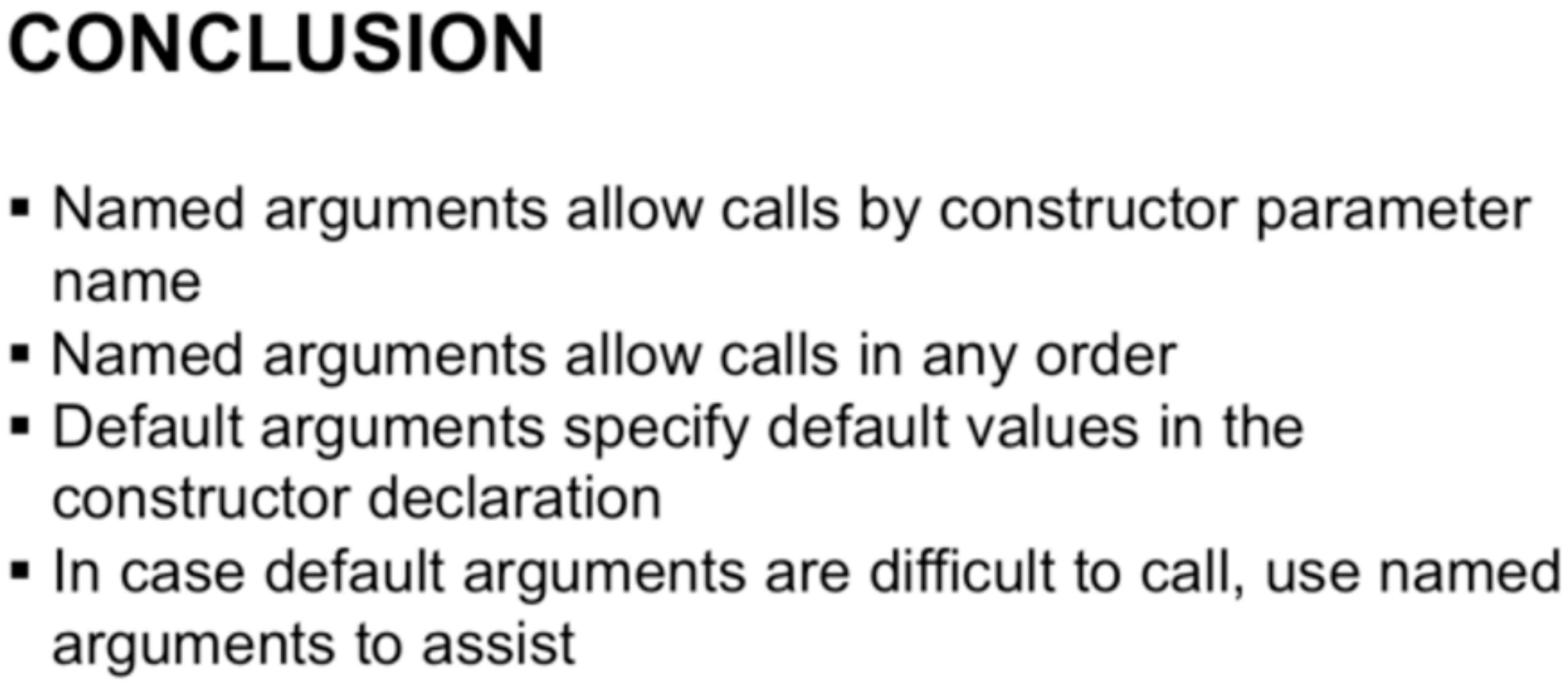
Ancillary Constructors



CONCLUSION

- Ancillary Constructors look like methods named `this`
- Primary Constructors are designed for all information up front
- Ancillary Constructors need to find a way to invoke the primary constructor
- Call another constructor by invoking `this(...)`
- If an ancillary constructor is multi-lined, the first line must be a call to `this(...)`

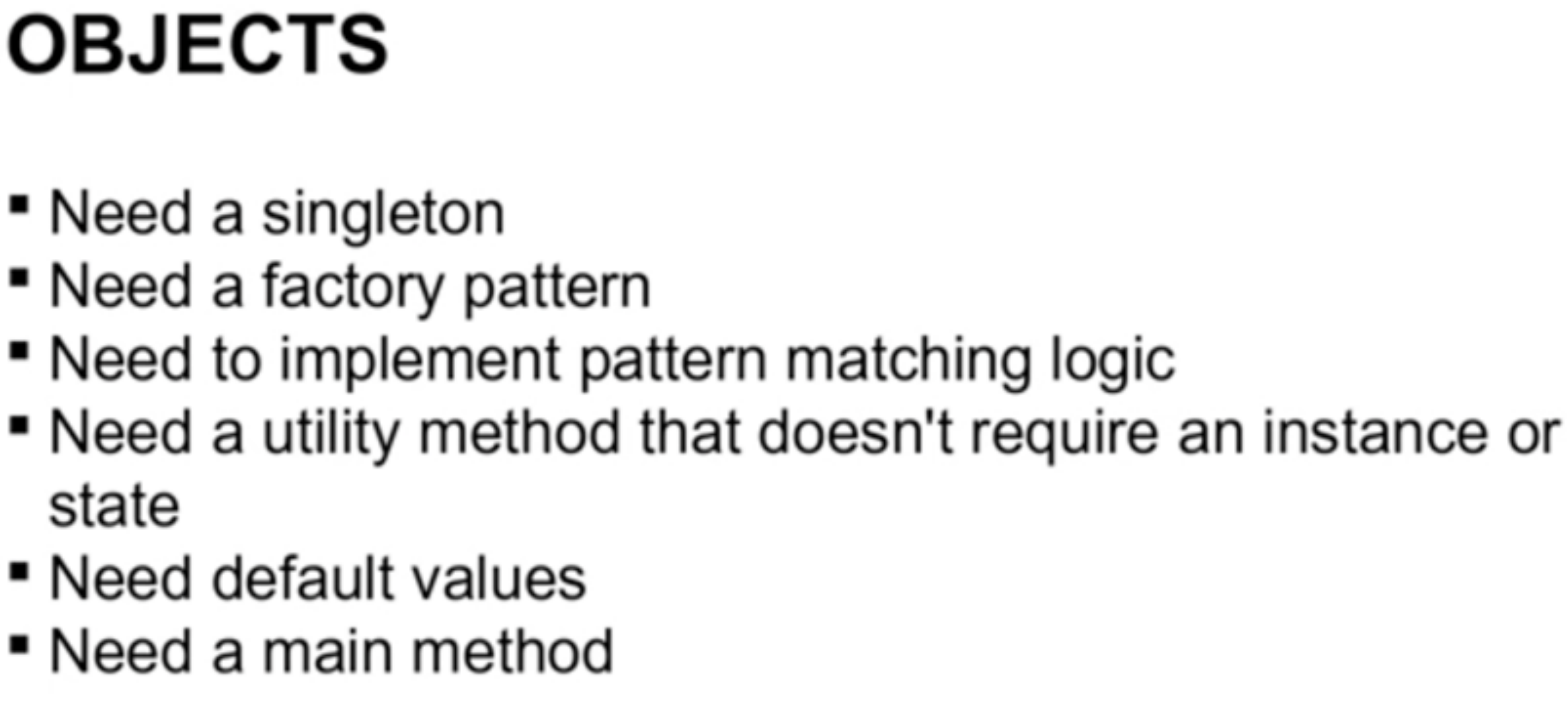
Constructor Named and Default Arguments



CONCLUSION

- Named arguments allow calls by constructor parameter name
- Named arguments allow calls in any order
- Default arguments specify default values in the constructor declaration
- In case default arguments are difficult to call, use named arguments to assist

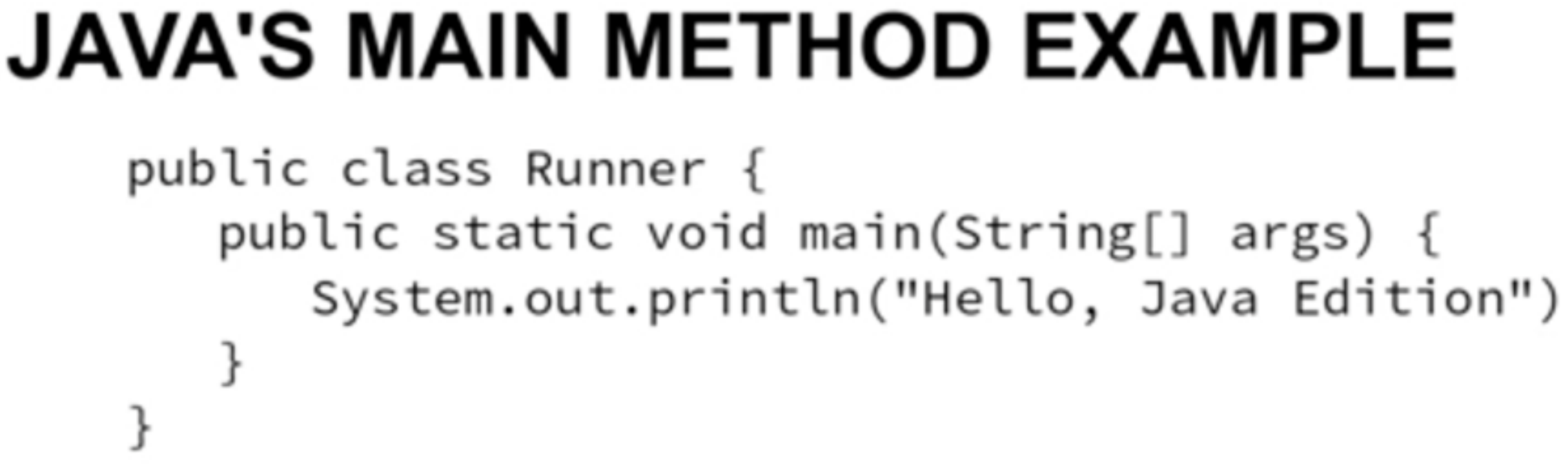
Singleton Objects



OBJECTS

- Need a singleton
- Need a factory pattern
- Need to implement pattern matching logic
- Need a utility method that doesn't require an instance or state
- Need default values
- Need a main method

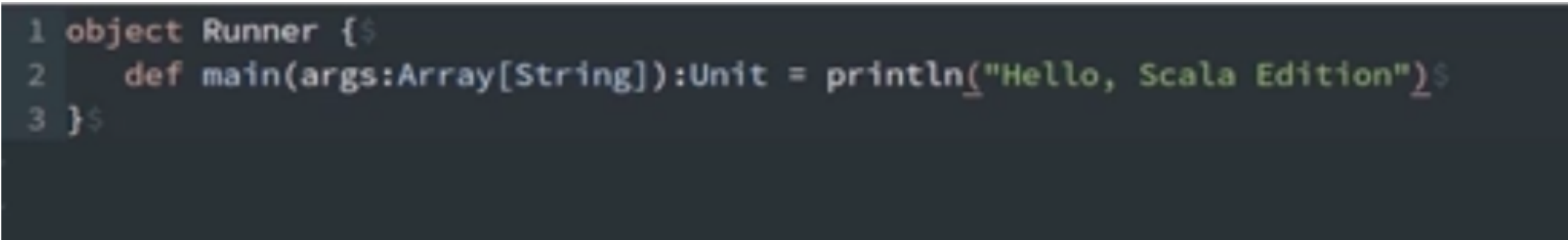
Main Method



JAVA'S MAIN METHOD EXAMPLE

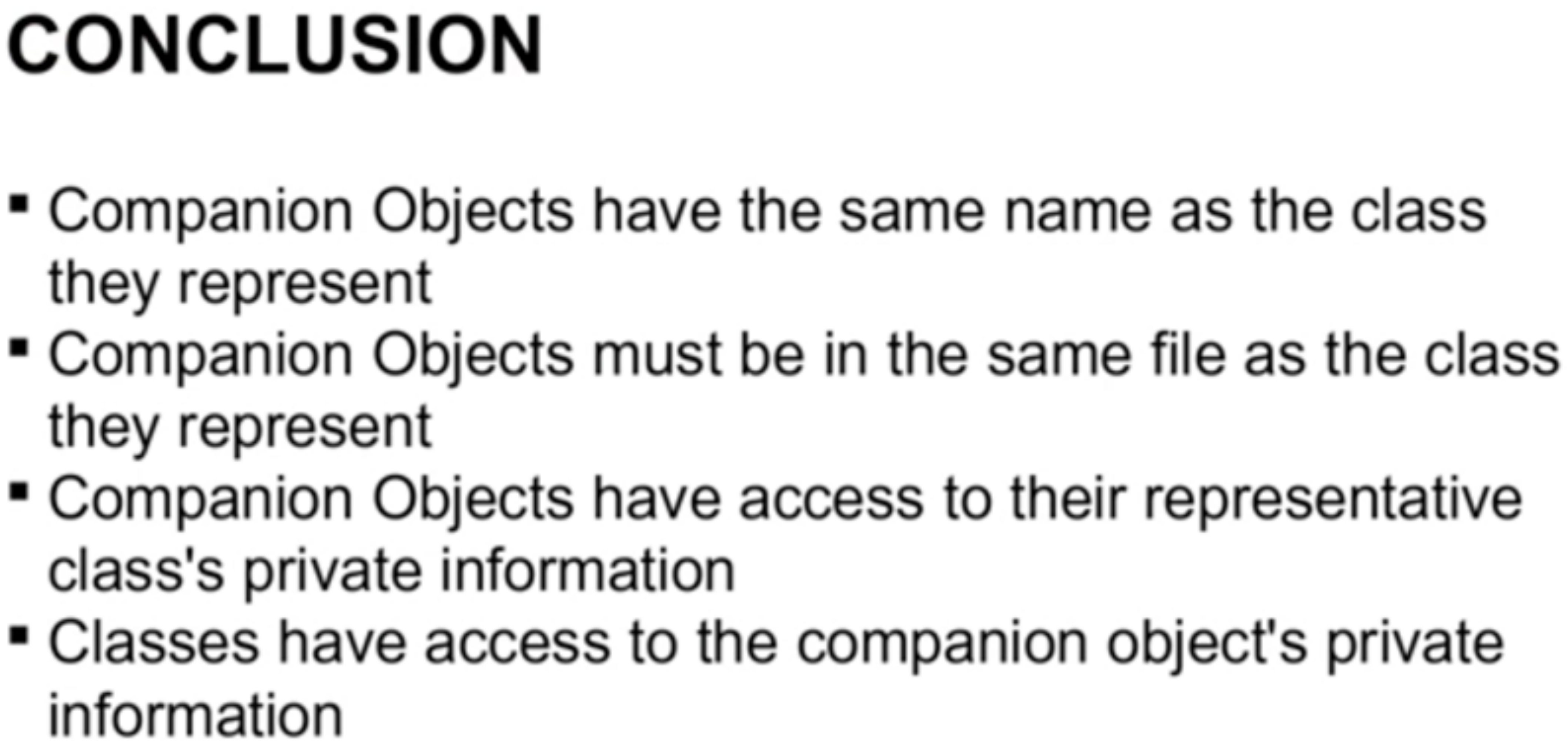
```
public class Runner {
   public static void main(String[] args) {
      System.out.println("Hello, Java Edition")
   }
}
```

SCALA’s MAIN METHOD



```
object Runner {
   def main(args:Array[String]):Unit = println("Hello, Scala Edition")
}
```

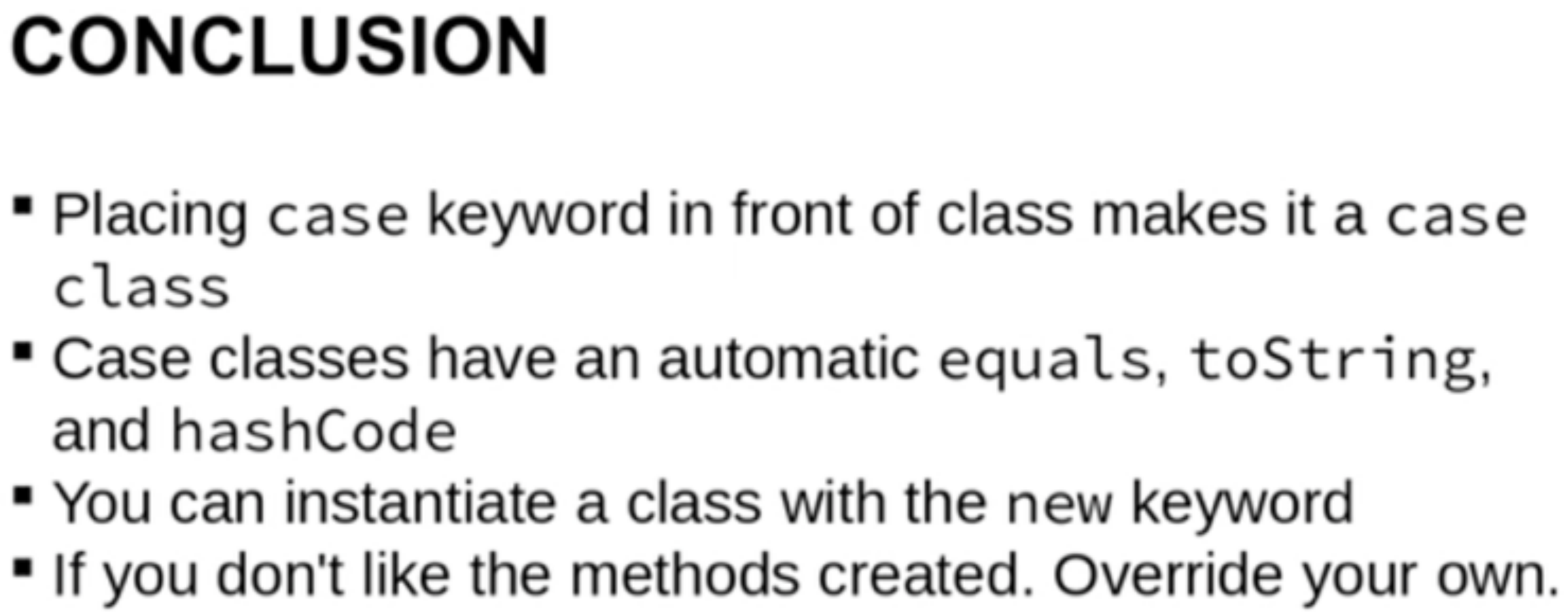
Companion Objects



CONCLUSION

- Companion Objects have the same name as the class they represent
- Companion Objects must be in the same file as the class they represent
- Companion Objects have access to their representative class's private information
- Classes have access to the companion object's private information

Case Classes

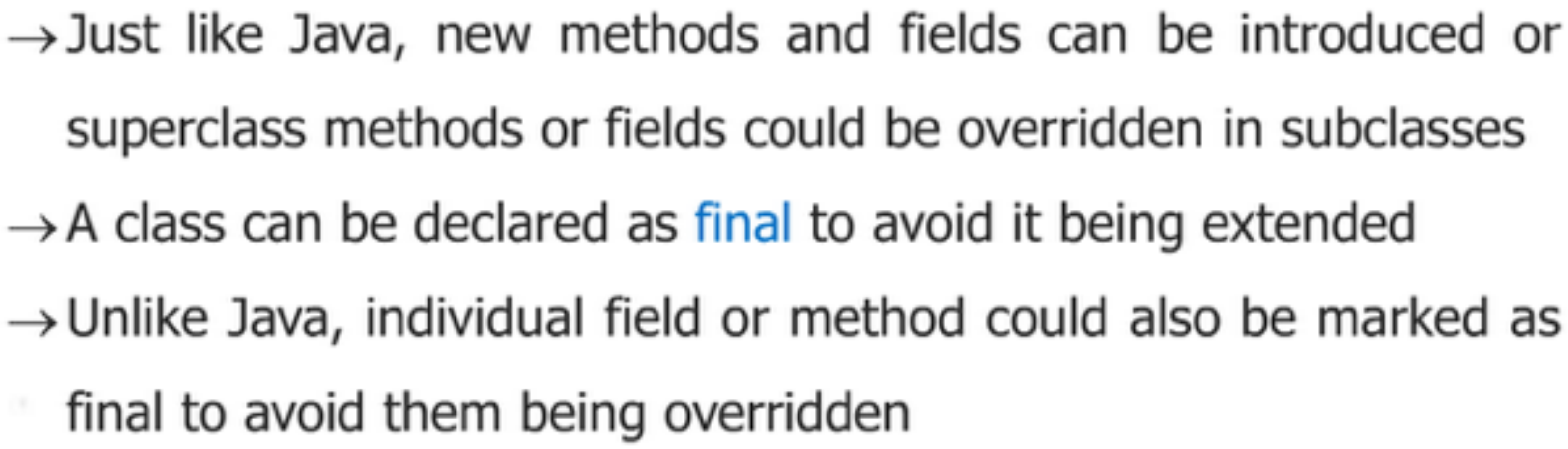


CONCLUSION

- Placing `case` keyword in front of class makes it a `case class`
- Case classes have an automatic `equals`, `toString`, and `hashCode`
- You can instantiate a class with the `new` keyword
- If you don't like the methods created. Override your own.

Extending Classes

- Extending a Class in Scala is similar to JAVA



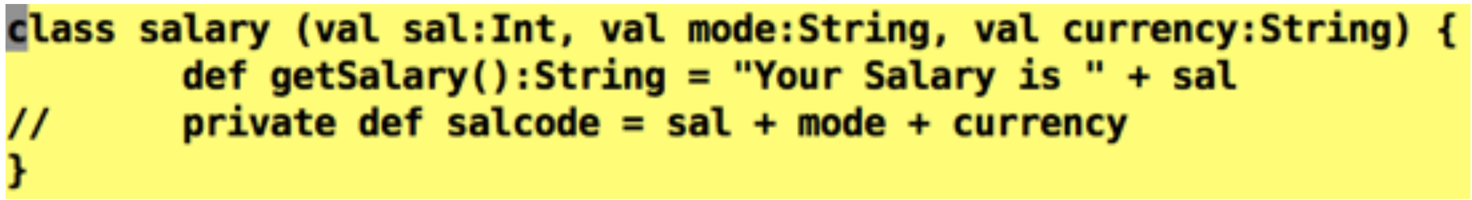
→Just like Java, new methods and fields can be introduced or superclass methods or fields could be overridden in subclasses

→A class can be declared as final to avoid it being extended

→Unlike Java, individual field or method could also be marked as final to avoid them being overridden

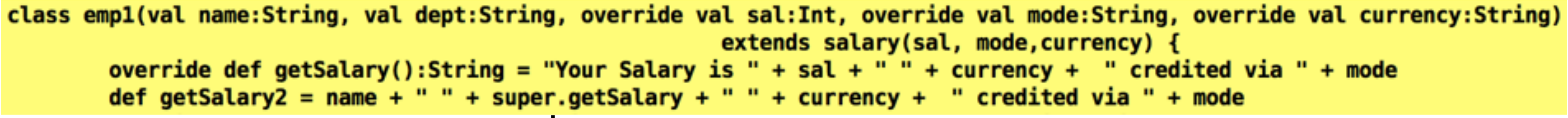
Extending Classes

Base Class



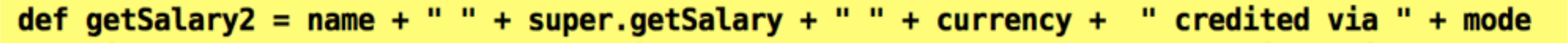
```
class salary (val sal:Int, val mode:String, val currency:String) {
        def getSalary():String = "Your Salary is " + sal
//      private def salcode = sal + mode + currency
}
```

Inherited by below



```
class emp1(val name:String, val dept:String, override val sal:Int, override val mode:String, override val currency:String)
                                                     extends salary(sal, mode,currency) {
        override def getSalary():String = "Your Salary is " + sal + " " + currency +  " credited via " + mode
        def getSalary2 = name + " " + super.getSalary + " " + currency +  " credited via " + mode
```

You can refer any method of super class as below

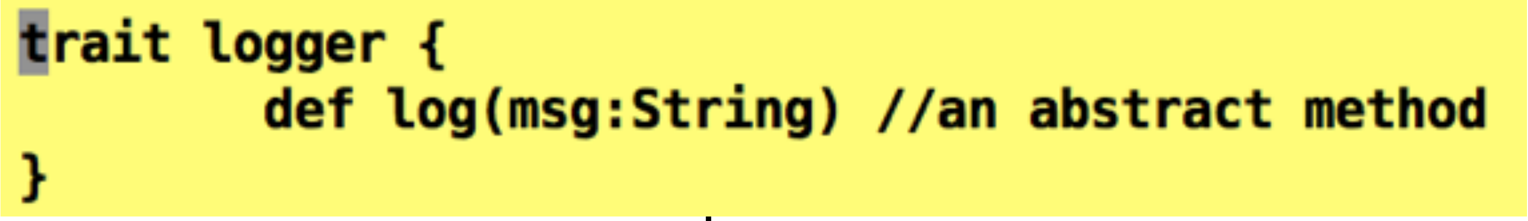


```
def getSalary2 = name + " " + super.getSalary + " " + currency +  " credited via " + mode
```

- Inherited class inherits all **NON-PRIVATE** members of the base class
- SCALA allows inheritance from one class only
- You can still refer the method from super class with the keyword **super**

Traits

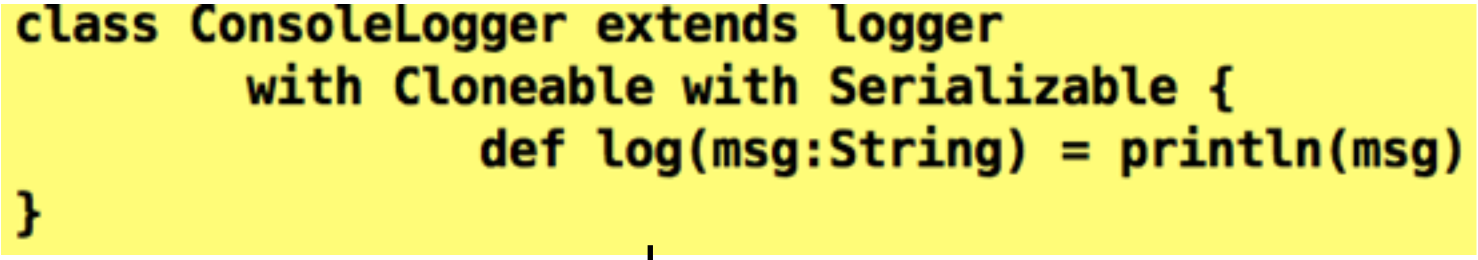
Trait1



```
trait logger {
        def log(msg:String) //an abstract method
}
```

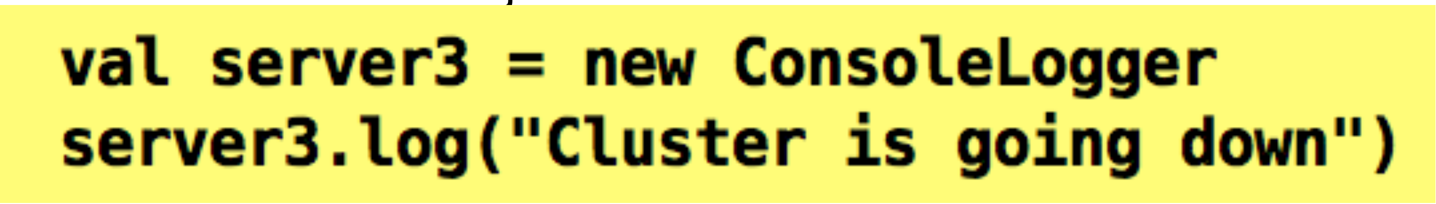
Extends

Class1



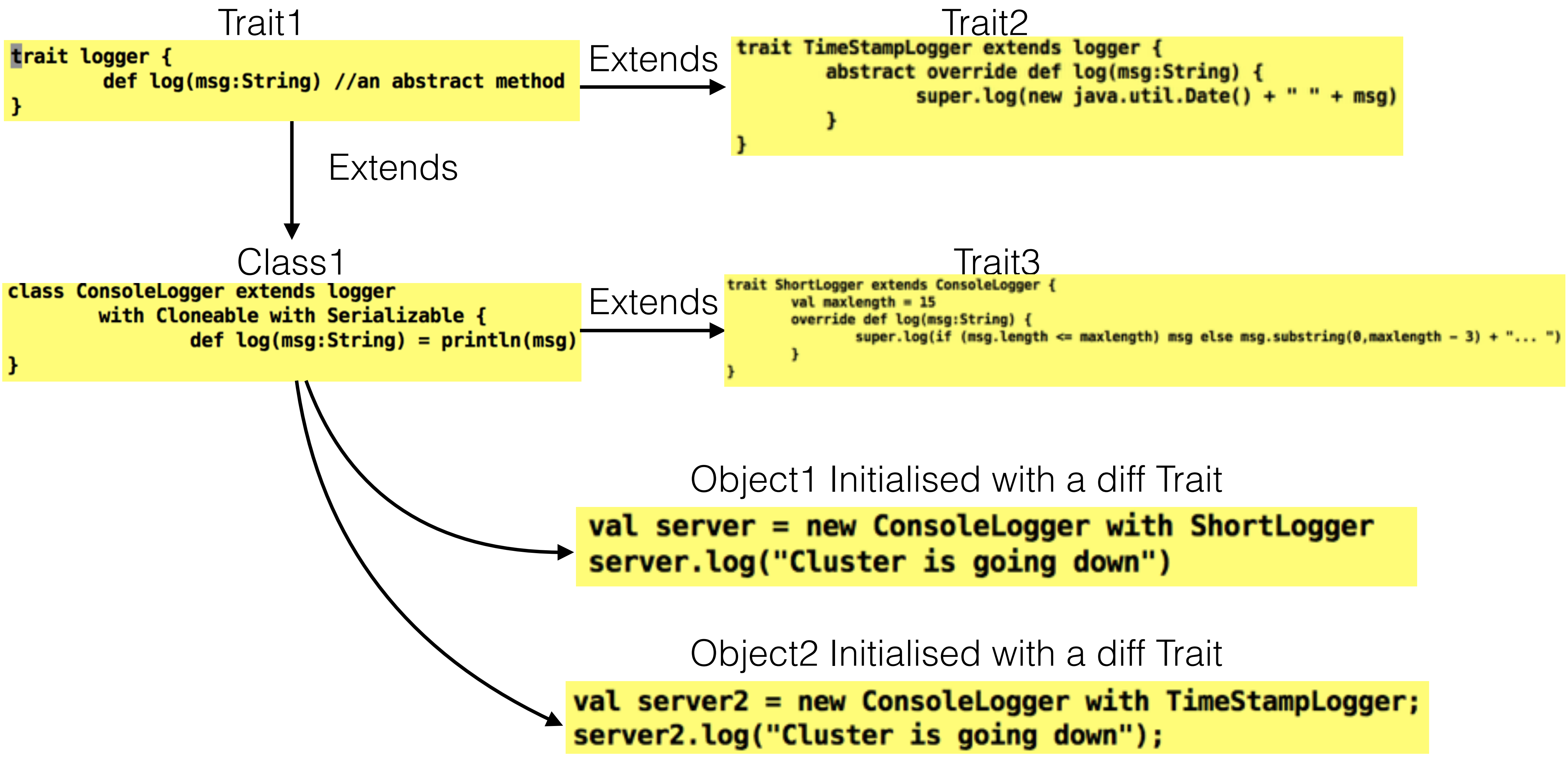
```
class ConsoleLogger extends logger
        with Cloneable with Serializable {
                def log(msg:String) = println(msg)
}
```

Object Initialised



```
val server3 = new ConsoleLogger
server3.log("Cluster is going down")
```

Multi-Layered Traits



Trait1

```
trait logger {
        def log(msg:String) //an abstract method
}
```

Extends

Trait2

```
trait TimeStampLogger extends logger {
        abstract override def log(msg:String) {
                super.log(new java.util.Date() + " " + msg)
        }
}
```

Extends

Class1

```
class ConsoleLogger extends logger
        with Cloneable with Serializable {
                def log(msg:String) = println(msg)
}
```

Extends

Trait3

```
trait ShortLogger extends ConsoleLogger {
        val maxlength = 15
        override def log(msg:String) {
                super.log(if (msg.length <= maxlength) msg else msg.substring(0,maxlength - 3) + "... ")
        }
}
```

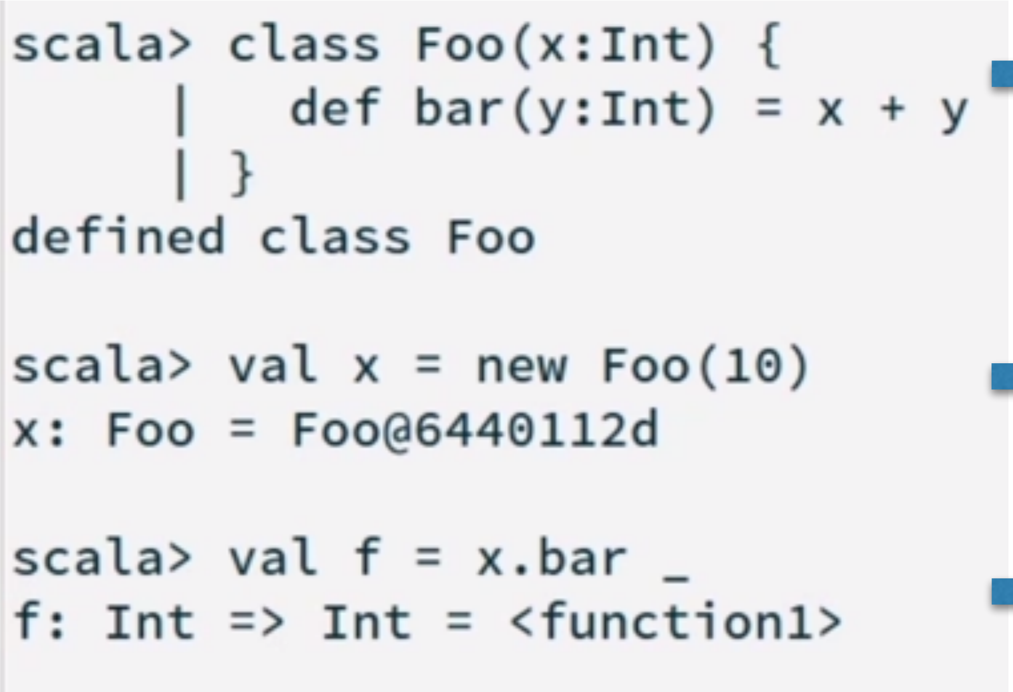
Object1 Initialised with a diff Trait

```
val server = new ConsoleLogger with ShortLogger
server.log("Cluster is going down")
```

Object2 Initialised with a diff Trait

```
val server2 = new ConsoleLogger with TimeStampLogger;
server2.log("Cluster is going down");
```

Converting methods into Functions



```
scala> class Foo(x:Int) {
     |   def bar(y:Int) = x + y
     | }
defined class Foo
```

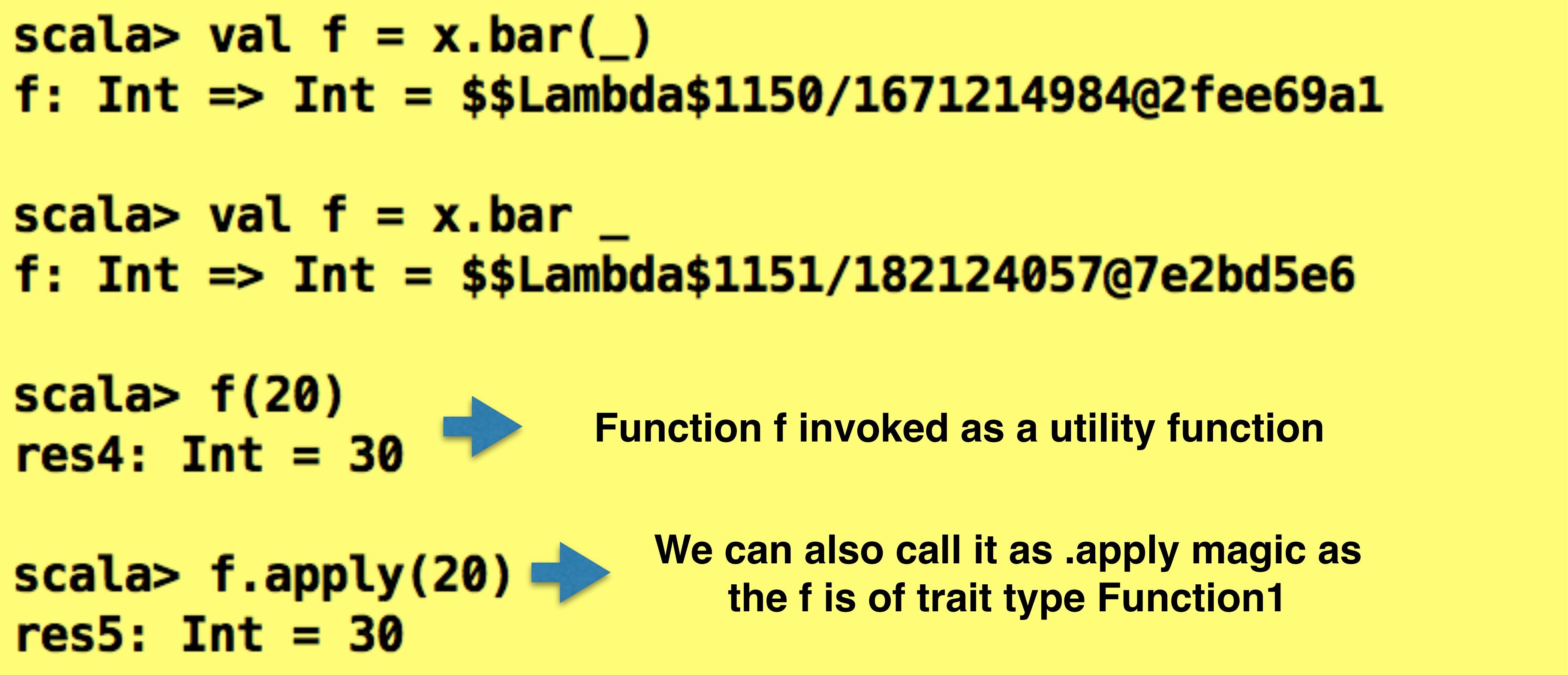
Class declared with a function of type Int => Int

```
scala> val x = new Foo(10)
x: Foo = Foo@6440112d
```

Object Declared on the class

```
scala> val f = x.bar _
f: Int => Int = <function1>
```

The METHOD x.bar converted into a function f



```
scala> val f = x.bar(_)
f: Int => Int = $$Lambda$1150/1671214984@2fee69a1

scala> val f = x.bar _
f: Int => Int = $$Lambda$1151/182124057@7e2bd5e6

scala> f(20)
res4: Int = 30
```

Function f invoked as a utility function

```
scala> f.apply(20)
res5: Int = 30
```

We can also call it as .apply magic as the f is of trait type Function1