**SCALA OOP:**

**Regole:**

Scala supports single inheritance, not multiple inheritance.   
A child (or derived) class can have one and only one parent (or base) class. The sole exception is the root of the Scala class hierarchy, **Any**, which has no parent.

**Classi (wrapper su java primitivi):**

Sono inizializzate sempre tramite literal.  
Non esiste per esempio un new Int(1);

**Constructors in Scala**

Scala distinguishes between a primary constructor and zero or more auxiliary constructors.

In Scala, the primary constructor is the entire body of the class.   
Any parameters that the constructor requires are listed after the class name.  
  
Cio che si trova nel body della classe, e non consiste in definizioni (membri classe) viene eseguito. (test con dei print)

Costruttori ausiliari vengono create con

def this (params)

To avoid infinite recursion, Scala requires each auxiliary constructor to invoke another

constructor defined before it (see [ScalaSpec2009]). The constructor invoked may be

either another auxiliary constructor or the primary constructor, and it must be the first

statement in the auxiliary constructor’s body. Additional processing can occur after

this call, such as the warning message printed in our example.

**VANTAGGI SVANTAGGI COSTRUTTORI SCALA RISPETTO A JAVA**

There are a few advantages of Scala’s constraints on constructors:

*Elimination of duplication*

Because auxiliary constructors invoke the primary constructor, potential duplication

of construction logic is largely eliminated.

*Code size reduction*

As shown in the examples, when one or more of the primary constructor parameters

is declared as a val or a var, Scala automatically generates a field, the appropriate

accessor methods (unless they are declared private), and the initialization

logic for when instances are created.

There is also at least one disadvantage of Scala’s constraints on constructors:

*Less flexibility*

Sometimes it’s just not convenient to have one constructor body that all constructors

are forced to use. However, we find these circumstances to be rare. In such

cases, it may simply be that the class has too many responsibilities and it should

be refactored into smaller classes.

**Campi (parametri passati al costruttore): Incapsulamento**

Because each parameter is declared as a val, the compiler generates a private field corresponding to each parameter (a different internal name is used), along with a public reader method that has the same name as the parameter.

If a parameter has the var keyword, a public writer method is also generated with the parameter’s name as a prefix, followed by \_=.   
For example, if label were declared as a var, the writer method would be named label\_= and it would take a single argument of type String.

*Uniform Access Principle*.  
Override di questo metodo di default permette l’incapsulamento (setters)

There are times when you don’t want the accessor methods to be generated automatically.

In other words, you want the field to be *private*. Add the private keyword before

the val or var keyword, and the accessor methods won’t be generated

**Ereditarieta:**

If you don’t extend a parent class, the default parent is AnyRef, a direct child class of

Any.

Chiamata a costruttore parent:

The primary constructor in a derived class must invoke one of the parent class constructors,

either the primary constructor or an auxiliary constructor

class <SubClass> (params declared)   
extends <ParentClass> (commonParams from declared in subclass args)

**Override metodi**

override def metodo() (se manca override non compila)

Public Visibility

Any declaration without a visibility keyword is “public,” meaning it is visible everywhere.

There is no public keyword in Scala.

**Possibilita’ di assegnare private e protected ad uno scope.**

Es private [package]

**Scopes:**package (estende il concetto al package)  
class (quello di default)  
this (restringe alla istanza in esecuzione ad esempio sul metodo equals)

The private[this] members are only visible to the same instance. An instance of the same class can’t see private[this] members of another instance, so the equalFields method won’t parse.

Otherwise, the visibility of class members is the same as private without a scope specifier.

**Abstract fields:**

Possibile definirli in classi abstract o Traits.  
Sono var o val senza inizializzazione.

*Uniform Access Principle*:  
Membri (var, val e metodi (def) sono nello stesso namespace.)  
E’ possibile invocare un metodo attraverso il nome del membro pubblico.  
Se ha parametri di input

What about overriding a parameterless method with a var, or overriding a val or var

with a method? These are not permitted because they can’t match the behaviors of the

things they are overriding.

If you attempt to use a var to override a parameterless method, you get an error that

the writer method, override name\_=, is not overriding anything.

This would also be inconsistent with a philosophical goal of functional programming, that a method that

takes no parameters should always return the same result.   
To do otherwise would require side effects in the implementation, which functional programming tries to

avoid, for reasons we will examine in Chapter 8.   
Because a var is changeable, the noparameter “method” defined in the parent type would no longer return the same result consistently.

If you could override a val with a method, there would be no way for Scala to guarantee

that the method would always return the same value, consistent with val semantics.

That issue doesn’t exist with a var, of course, but you would have to override the var

with two methods, a reader and a writer. The Scala compiler doesn’t support that

substitution.

**Companion class:**

**Object e class nello stesso file.**

Recall that fields and methods defined in objects serve the role that class “static” fields

and methods serve in languages like Java. When object-based fields and methods are

closely associated with a particular class, they are normally defined in a *companion*

*object*.

First, recall that if a class (or a type referring to a class) and an object are declared in

the same file, in the same package, and with the same name, they are called a *companion*

*class* (or *companion type*) and a *companion object*, respectively.

There is no namespace collision when the name is reused in this way, because Scala

stores the class name in the type namespace, while it stores the object name in the term

namespace

The two most interesting methods frequently defined in a companion object are

apply and unapply.

**Apply e unapply:** Il companion Object puo’ definire dei metodi apply(params) e unapply(params)  
  
 **apply()** permette di usare la classe come una funzione.

La funzione apply verra’ invocata con Classe(argsDiApply) e restituira’ il tipo di

ritorno definite in apply.  
  
per cui var x = Classe(args) = CompanionObject.apply(args)

**Unapply** viene usato come extractor dell oggetto sovrascrivendo il comportamento

degli Extractor usati nel pattern match

per cui in un match su un oggetto di tipo Classe

case Classe(valore) = CompanionObject.unapply(valore)   
Se il metodo (che restituisce un Optional) restituisce un Some allora il match riuscira’, altrimenti col None, fallira’.

**Apply: (metodo factory)**

When an instance of a class is followed by parentheses with a list of zero or more parameters, the compiler invokes the apply method for that instance. This is true for an object with a defined apply method (such as a companion object), as well as an instance of a class that defines an apply method.

In the case of an object, apply is conventionally used as a *factory* method, returning a new instance.

**Best practice:**

If there are several alternative constructors for a class and it also has a

companion object, consider defining fewer constructors on the class and

defining several overloaded apply methods on the companion object to

handle the variations.

**Si puo fare anche se non capisco con quali vantaggi:**

However, apply is not limited to instantiating the companion class. It could instead

return an instance of a subclass of the companion class

**Unapply**

The name unapply suggests that it does the “opposite” operation of apply.   
Indeed, it is used to extract the constituent parts of an instance.   
Pattern matching uses this feature extensively.

Hence, unapply is often defined in companion objects and is used to extract

the field values from instances of the corresponding companion types. For this reason,

unapply methods are called *extractors*.

**CASE CLASSES:**

Adding the case keyword causes the compiler to add a number of useful features automatically.

The keyword suggests an association with case expressions in pattern

matching.   
Indeed, they are particularly well suited for that application, as we will see

First, the compiler automatically converts the constructor arguments into immutable

fields (vals). The val keyword is optional. If you want mutable fields, use the var keyword.

So, our constructor argument lists are now shorter.

Second, the compiler automatically implements equals, hashCode, and toString methods

to the class, which use the fields specified as constructor arguments. So, we no

longer need our own toString methods

Even outside of case expressions, automatic generation of these three methods is very

convenient for simple, “structural” classes, i.e., classes that contain relatively simple

fields and behaviors.

Third, when the case keyword is used, the compiler automatically creates a *companion*

*object* with an apply factory method that takes the same arguments as the *primary*

constructor. The previous example used the appropriate apply methods to create the

Points, the different Shapes, and also the List itself. That’s why we don’t need new;

we’re actually calling apply(x,y) in the Point companion object, for example.

You can have *secondary* constructors in case classes, but there will be

no overloaded apply method generated that has the same argument list.

You’ll have to use new to create instances with those constructors.

The companion object also gets an unapply extractor method, which extracts all the

fields of an instance in an elegant fashion.

**COPY (in case classes)**

In Scala version 2.8, another instance method is automatically generated, called copy.

This method is useful when you want to make a new instance of a case class that is

identical to another instance with a few fields changed.   
(vedi demo copy)

**Case Class Inheritance (evitare su abstract)**

Non e’ in genere una buona idea avere case classes abstract , perche’ cio che viene generato dal compilatore facilmente andra’ in conflitto con quello che viene scritto.

Tipicamente mancheranno gli override.

**Equals**

**The equals Method**

The equals method tests for *value* equality.   
That is, obj1 equals obj2 is true if both obj1 and obj2 have the same value.   
They do not need to refer to the same instance.

Hence, equals behaves like the equals method in Java

**The == and != Methods (delegano al metodo equals eventualmente overriden)**

While == is an operator in many languages, it is a method in Scala, defined as final in

Any.   
It tests for *value* equality, like equals.   
That is, obj1 == obj2 is true if both obj1 and obj2 have the same value.   
In fact, == delegates to equals.   
Here is part of the Scaladoc entry for Any.==: o == arg0 is the same as o.equals(arg0).

Here is the corresponding part of the Scaladoc entry for AnyRef.==:

o == arg0 is the same as if (o eq null) arg0 eq null else o.equals(arg0).

As you would expect, != is the negation, i.e., it is equivalent to !(obj1 == obj2).

Since == and != are declared final in Any, you can’t override them, but you don’t need to, since they delegate to equals.

In Java, C++, and C#, the == operator tests for *reference*, not *value* equality.

**make sure to remember that in Scala, == is testing for value equality.**

**The ne and eq Methods (equivalente di Object.equals java)**

The eq method tests for *reference* equality.   
That is, obj1 eq obj2 is true if both obj1 and obj2 point to the same location in memory.   
These methods are only defined for AnyRef.

Hence, eq behaves like the == operator in Java, C++, and C#, but not == in Ruby.

The ne method is the negation of eq, i.e., it is equivalent to !(obj1 eq obj2).

**Array Equality and the sameElements Method**

Comparing the contents of two Arrays doesn’t have an obvious result in Scala:

scala> Array(1, 2) == Array(1, 2)

res0: Boolean = false

That’s a surprise! Thankfully, there’s a simple solution in the form of the

**sameElements** method:

scala> Array(1, 2).sameElements(Array(1, 2))

res1: Boolean = true

Remember to use sameElements when you want to test if two Arrays contain

the same elements.

**Sospesi:**

* Costruttori ausiliari per campi e mantenere l incapsulamento.
* Riutilizzo codice costruttore.