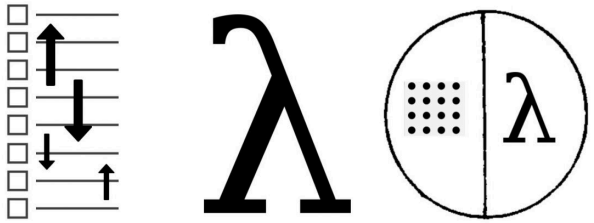
Fundusze Europejskie
Wiedza Edukacja Rozwój

Politechnika Wrocławska

Unia Europejska
Europejski Fundusz Społeczny

„ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej”

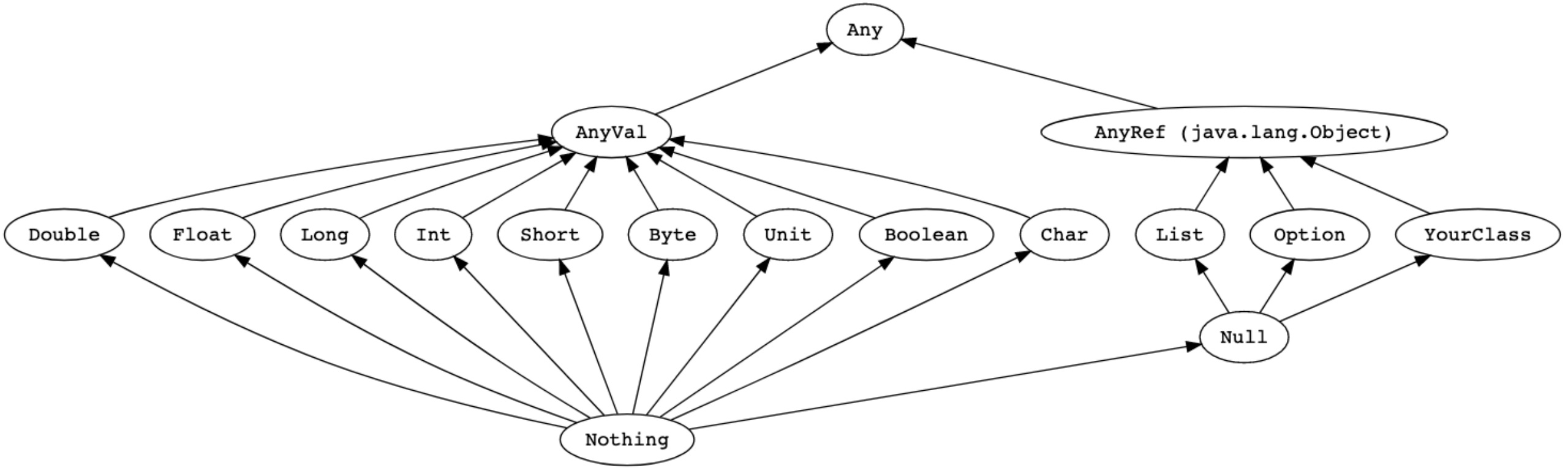


Programming paradigms

L09: OOP - Scala

by Michał Szczepanik

Type hierarchy



Any

Any is the supertype of all types, also called the top type.

It defines certain universal methods such as:

- equals,
- hashCode,
- toString.

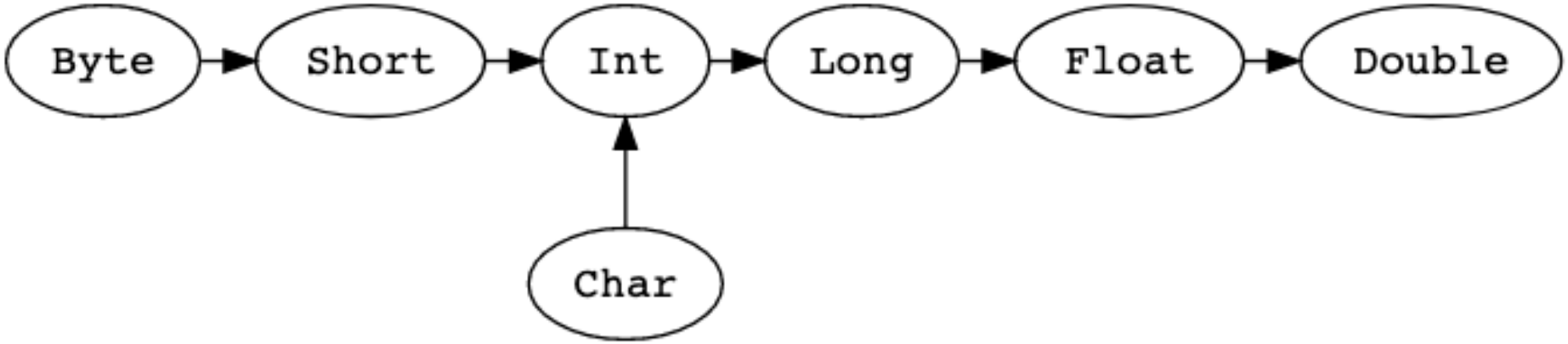
Any has two direct subclasses: AnyVal and AnyRef.

AnyVal

AnyVal represents value types. There are nine predefined value types and they are non-nullable:
Double, Float, Long, Int, Short, Byte, Char, Unit, and *Boolean*.

Unit is a value type which carries no meaningful information.
There is exactly one instance of Unit which can be declared literally like so: (). All functions must return something so sometimes Unit is a useful return type.

Type casting




```
val x: Long = 987654321
val y: Float = x // 9.8765434E8
val face: Char = '☺'
val number: Int = face // 9786
```

AnyRef

AnyRef represents reference types. All non-value types are defined as reference types. Every user-defined type in Scala is a subtype of AnyRef. If Scala is used in the context of a Java runtime environment, AnyRef corresponds to java.lang.Object.

Object Equality in Scala

- The *equals* Method
- The == and != Methods
- The *ne* and *eq* Methods

Object Equality in Scala

- **equals Method:** The equals method used to tests value equality. if x equals y is true if both x and y have the same value. They do not need to refer to the identical instance. Hence, the equals method in Java and equals method in Scala behaves same.
- **The == and != Methods:** While == is an operator in several languages, Scala reserved The == equality for the natural equality of every type. it's a method in Scala, defined as final in Any. value equality will be tested by this.
- **ne and eq Methods:** Reference equality will be tested by eq method. Here, x eq y is true if both x and y point to the same location in memory or x and y reference the same object. These methods are only defined for AnyRef.

Pros and Cons of objects

Immutable object trade-offs [M.Odersky, L.Spoon, B.Venners, Programming in Scala]

Immutable objects offer several advantages over mutable objects, and one potential disadvantage.

- First, immutable objects are often easier to reason about than mutable ones, because they do not have complex state spaces that change over time.
- Second, you can pass immutable objects around quite freely, whereas you may need to make defensive copies of mutable objects before passing them to other code.
- Third, there is no way for two threads concurrently accessing an immutable to corrupt its state once it has been properly constructed, because no thread can change the state of an immutable.
- Fourth, immutable objects make safe hash table keys. If a mutable object is mutated after it is placed into a HashSet, for example, that object may not be found the next time you look into the HashSet.

The main disadvantage of immutable objects is that they sometime require that a large object graph be copied where otherwise an update could be done in place. In some cases this can be awkward to express and might also cause a performance bottleneck. As a result, it is not uncommon for libraries to provide mutable alternatives to immutable classes. For example, class StringBuilder is a mutable alternative to the immutable String.

Traits

Traits are used to share interfaces and fields between classes.

```
trait Iterator[A] {
    def hasNext: Boolean
    def next(): A
}
```

Subtyping

```
trait Pet {
     val name: String
}

class Cat(val name: String) extends Pet
class Dog(val name: String) extends Pet

val dog = new Dog("Harry")
val cat = new Cat("Sally")
```

Class

A class may hold the following members:

- Data members
- Member methods
- Constructors
- Blocks
- Nested classes
- Information about the superclass

Declaration of class

- **Keyword class:** A class keyword is used to declare the type class.
- **Class name:** The name should begin with a initial letter (capitalized by convention).
- **Superclass(if any):**The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
- **Traits(if any):** A comma-separated list of traits implemented by the class, if any, preceded by the keyword extends. A class can implement more than one trait.
- **Body:** The class body is surrounded by { } (curly braces).

Declaration of class

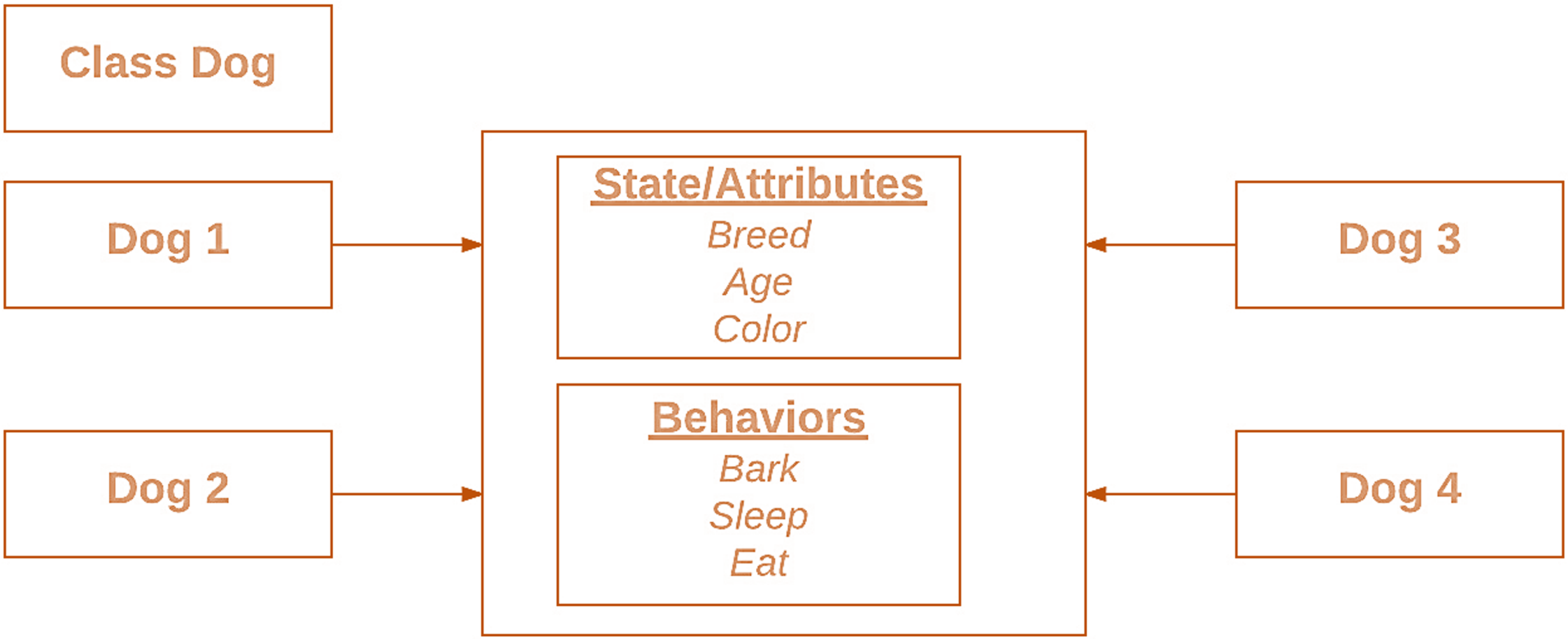
```
class Class_name{
      // methods and feilds
}
```

Note: The default modifier of the class is public.

Object

It is a basic unit of Object Oriented Programming and represents the real-life entities. A typical Scala program creates many objects, which as you know, interact by invoking methods. An object consists of :

- **State:** It is represented by attributes of an object. It also reflects the properties of an object.
- **Behavior:** It is represented by methods of an object. It also reflects the response of an object with other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.


Class Dog
Dog 1
Dog 2
State/Attributes
Breed
Age
Color
Behaviors
Bark
Sleep
Eat
Dog 3
Dog 4

```
class Dog(name:String, breed:String, age:Int, color:String )
{
        println("My name is:" + name + " my breed is:" + breed);
        println("I am: " + age + " and my color is :" + color);

}
object Main
{

    // Main method
    def main(args: Array[String])
    {

        // Class object
        var obj = new Dog("tuffy", "papillon", 5, "white");
    }
}
```

Extending a Class in Scala

Extending a class in Scala user can design an inherited class. To extend a class in Scala we use extends keyword. there are two restrictions to extend a class in Scala :

- To override method in scala override keyword is required.
- Only the primary constructor can pass parameters to the base constructor

```
class base_class_name extends derived_class_name {
// Methods and fields
}
```

Encapsulation

```
class Encapsulation(hidden: Any, val readable: Any, var settable: Any)
```

Scala access modifiers (scopes)

- Object-private scope
- Private
- Package
- Package-specific
- Public

Object-private scope

The most restrictive access is to mark a method as “object-private.” When you do this, the method is available only to the current instance of the current object. Other instances of the same class cannot access the method.

```
class Foo {
  private[this] def isFoo = true
  def doFoo(other: Foo) {
    if (other.isFoo) { // this line won't compile
      // ...
    }
  }
}
```

Private scope

A slightly less restrictive access is to mark a method private, which makes the method available to (a) the current class and (b) other instances of the current class. This is the same as marking a method private in Java.

```
class Foo {
  private def isFoo = true
  def doFoo(other: Foo) {
     if (other.isFoo) { // this now compiles
       // ...
     }
   }
}
```

Protected scope

Marking a method protected makes the method available to subclasses. The meaning of protected is slightly different in Scala than in Java. In Java, protected methods can be accessed by other classes in the same package, but this isn’t true in Scala.

Package scope

To make a method available to all members of the current package — what would be called “package scope” in Java — mark the method as being private to the current package with
the *private[packageName]* syntax.

Package-level control

Beyond making a method available to classes in the current package, Scala gives you more control and lets you make a method available at different levels in a class hierarchy.

```
package com.acme.coolapp.model {
 class Foo {
  private[model] def doX {}
  private[coolapp] def doY {}
  private[acme] def doZ {}
 }
}
```

Under JVM

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

Compiled from "Person.scala"
public class Person {
  private final java.lang.String name; // field
  public java.lang.String name(); // getter method
  public Person(java.lang.String); // constructor
}
```

Under JVM

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

```
Compiled from "Person.scala"
public class Person {
  private java.lang.String name; // field
  public java.lang.String name(); // getter method
  public void name_$eq(java.lang.String); // setter method
  public Person(java.lang.String); // constructor
}
```

Singleton

An object is a class that has exactly one instance. It is created lazily when it is referenced, like a lazy val.

```
object Logger { def info(message: String): Unit =
     println(s"INFO: $message") }
```

Companion objects

An object with the same name as a class is called a *companion object*. Conversely, the class is the object's companion class.

A companion class or object can access the private members of its companion. Use a companion object for methods and values which are not specific to instances of the companion class.

Companion objects

```
class Email(val username: String, val domainName: String)
object Email {
  def fromString(emailString: String): Option[Email] = {
    emailString.split('@') match {
      case Array(a, b) => Some(new Email(a, b))
      case _ => None
    }
  }
}
```

Companion objects (JVM)

static members in Java are modeled as ordinary members of a companion object in Scala.

When using a companion object from Java code, the members will be defined in a companion class with a static modifier. This is called *static forwarding*. It occurs even if you haven't defined a companion class yourself.

Refactoring (OOP)

https://blog.arkency.com/oop-refactoring-from-a-god-class-to-smaller-objects/

Thank you for your attention

michal.szczepanik@pwr.edu.pl