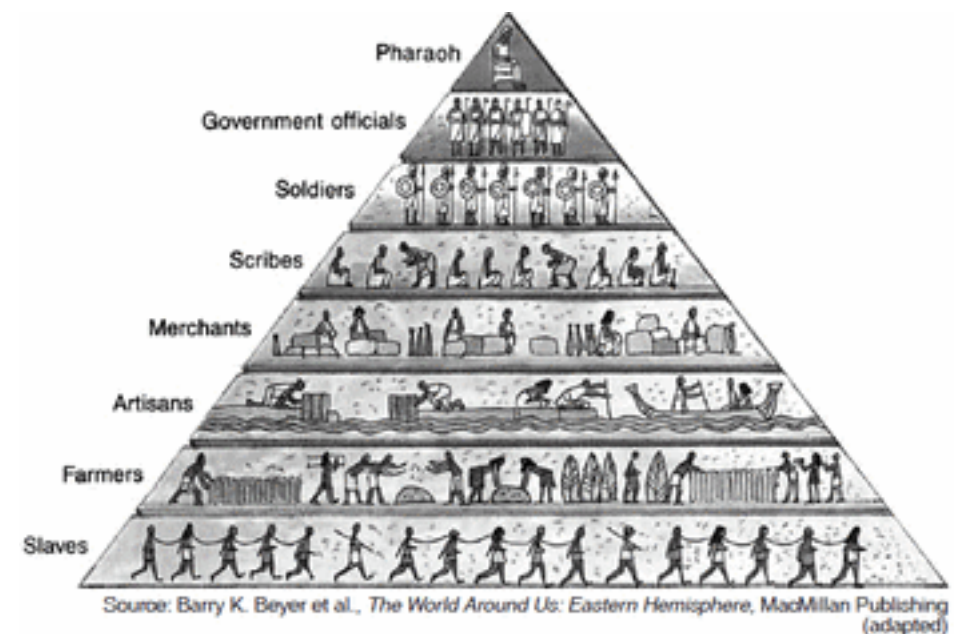


Class Hierarchies

Structure and organization



Based on the information in this illustration, which statement about the society of ancient Egypt is accurate?

Abstract classes

- Can not be instantiated!
- Use **abstract** keyword
- Use **extends** keyword to define a subclass
- The abstract class is a superclass of the realizations - baseclass
- Can have undefined methods
- The implementing class has to add method implementations

```
abstract class Animal(name: String)
```

Dynamic binding

- Dynamic method dispatch - which code will be called?
- In object oriented languages, the code invoked by method call depends on the runtime type of the object

```
empty contains 1
```

Overriding

- Redefine functionality
- Use the keyword **override**
- In Scala, **override** is mandatory

```
override def toString() = "."
```

Object definition

- Represents a singleton
- Singleton objects are values
- Reference - only one can exist
- Reference by name
- Can not have generic types (huh?? future lesson!)

```
object Empty extends IntSet {  
  def contains(n: Int) = false  
  def include(n: Int) = new NonEmpty(n, Empty, Empty)  
  override def toString() = "."  
}
```

Persistent data structures

- On change, the old structure still exists



What about multiple inheritance?

- We can build large class hierarchies
- Scala is a single inheritance language - a class can only have **one superclass**
- What if a class has several natural supertypes?
- What if we want to inherit code from several types?
- Example:
 - Platypus is a mammal, but lays eggs
 - Credit card is an editable card, but should also have some security settings



Packages

- Stay organized - use packages
- Fully qualified class name is **packagename.classname**
- Think of this as last name + first name!

```
package dk.lundogbendsen.scala.hello
object Hello{ ...}
//
dk.lundogbendsen.scala.hello.Hello
```


Import

- Use import to avoid typing the fully qualified class name
- Named import
- Wildcard import

```
import dk.lundogbendsen.hello.Hello // imports just Hello
import dk.lundogbendsen.hello.{Hello,Bye} // imports Hello and Bye
import dk.lundogbendsen.hello._ // wildcard imports all
```

Automatic imports

- Scala automatically imports members from the following packages
 - scala
 - java.lang
 - scala.Predef
- Checkout scaladocs at: www.scala-lang.org/api/current

```
Int      scala.Int
Boolean  scala.Boolean
Object   java.lang.Object
require  scala.Predef.require
assert   scala.Predef.assert
```



Traits in Scala

Composing functionality across class hierarchies

Class parameters

- You can define a class with parameters (primary constructor)
- Primary constructor parameters with `val` and `var` are public
- prefix parameters with: `var`, `val` or nothing
 - `var` - can be modified
 - `val` - is a value
 - nothing - private values to the class
- Defines at the same time parameters and fields of a class
- Parameters without `val` or `var` are private values, visible only within the class

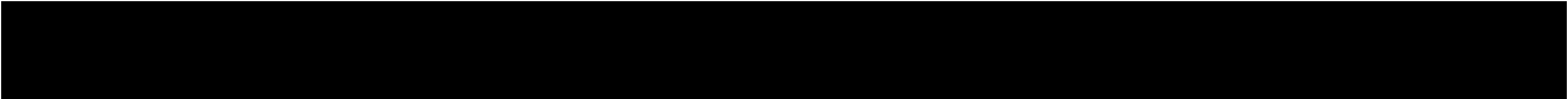
```
class Point(val x: Int, val y: Int)
val point = new Point(1, 2)
println( point.x)
point.x = 3    // <-- does not compile
```

Trait

- You can use traits, to inherit functionality from several places
- Classes, objects and traits can inherit from at most one class but arbitrary many traits
- Can't have constructor parameters
- A trait is declared like an abstract class, just with **trait** instead of **abstract class**
- When a class extends a trait, it uses the **extends** or **with** keywords
- When extending one trait, use **extends**
- When extending several traits, use **with**

```
trait BaseSoundPlayer {  
  def play  
  def close  
  def pause  
  def stop  
  def resume  
}  
class Mp3SoundPlayer extends BaseSoundPlayer { ...  
class Foo extends BaseClass with Trait1 with Trait2 { ...
```

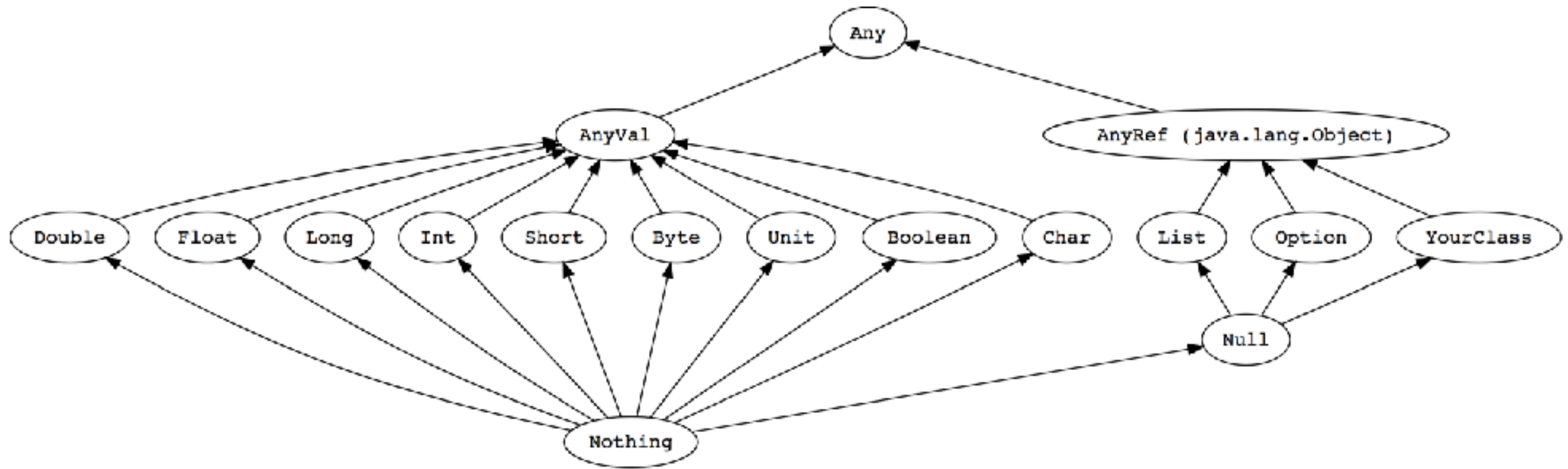
Class vs trait

- Traits resemble interfaces in Java, but are more powerful
 - Traits can contains fields and concrete methods.
 - Traits cannot have (value) parameters, only classes can.
- 

What is the difference?

- Abstract class vs Trait ???





Scala Type Hierarchy

Top Types

- In Scala, all values have a type, including numerical values and functions
- Any - the base of all types. Contains methods like ==, !=, equals, hashCode, toString
- AnyRef - The base type of all reference types; Alias of 'java.lang.Object'
- AnyVal - The base type of all primitive types
 - 9 predefined value types and they are non-nullable: Double, Float, Long, Int, Short, Byte, Char, Boolean, Unit
 - 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.

```
val list: List[Any] = List(  
  "a string",  
  732,    // an integer  
  'c',    // a character  
  true,   // a boolean value  
  () => "an anonymous function returning a string"  
)  
  
list.foreach(element => println(element))
```

Bottom types

- **Nothing** is a subtype of all types, also called the bottom type. There is no value that has type **Nothing**
 - signal abnormal termination
 - an element type of empty collection
- **Null** is a subtype of all reference types (i.e. any subtype of AnyRef).
 - It has a single value identified by the keyword literal null.
 - Null is provided mostly for interoperability with other JVM languages
 - should almost never be used in Scala code
 - Every reference class type also has null as a value. The type of null is Null.
 - Null is a subtype of every class that inherits from Object; it is incompatible with subtypes of AnyVal (non - nullable)

Nothing is useful for Exception handling

- exception handling is similar to Java
- The expression **throw Exc** aborts evaluation with the exception Exc
- The type of **throw Exc** is Nothing

```
def error(msg:String) = throw new Error(msg)
//> error: (msg: String)Nothing
```

Demo time!

Let's have a look at the types

Lab time!

class_hierarchies_06

