

What are the limitations on abstract types?

- They cannot create an instance of an abstract type.
- They cannot have an abstract types as a supertype of another type.
- Virtual classes may allow, but not currently supported in Scala.

Why do lazy values mesh better with functional programming?

If initialization of lazy `vals` produces side effects or depends on them. The order of initialization becomes important but cannot be guaranteed by lazy `val` author.

Give the inner class syntax.

Outer#Inner unlike Java's Outer.Inner.

'.' is reserved for objects in Scala.

How do you access the instance of your outer class from inside the inner class?

You can use the Java-style `Outer.this`.

Or use this aliasing:

```
class Outer { outer (immediately after opening brace of class or trait) =>
  class Inner {
    println(Outer.this eq outer) //true
  }
}
```


What are some features of Enumeration?

- Named values. Create with `Value("name")` instead of `Value`.

- retrieving `Int: Color.Red.id`

- retrieving value: `Color(I)`

What strategies does Scala provide for negating the risk of using uninitialized fields?

Pre-initialized fields, lazy `vals`.

What do abstract `vars` create?

They create abstract getters and setters with usual name.

A reassignable field itself is not created, so technically you can override getter and setter alone.

How do lazy vals eliminate the risk of using uninitialized values?

If a trait puts all initialization code behind lazy vals (remember, vals can have block definitions), there is effectively no constructor and no risk of any code using values until after the object is created. After all, the first access to an object only occurs after it is made, and only then is lazy code run.

What is a common use of abstract types?

They're used to narrow parameters of a superclass's abstract method in subtypes.

Instead the abstract method can be defined to take a parameter of an abstract type that has an appropriate upper bound. Now subtypes can define the abstract type appropriately for themselves.

What is the main advantage of abstract `vals` over abstract methods?

The guarantee to the client that the value will never change.

In other words, abstract `vals` constrain their legal implementations.

Describe the singleton type.

The type of a singleton object only has one instance - the singleton itself.

object O
O.type
is O's type.

Rarely useful, except to convince the compiler of something.
The compiler is reluctant to infer such a rare type.

Give a Scala shortcut to instantiating inner class that works outside the outer class.

```
val o = new Outer  
val i = new o.Inner
```


What can substitute for trait parameters
(which Scala does not allow) to create trait
instances?

Abstract vals combined with anonymous classes.

```
trait Rational {  
    val numer: Int  
    val denom: Int  
}  
  
new Rational {  
    val numer = 1; val denom = 2;  
}
```

Why are pre-initialized fields needed?

Only because traits have constructors but no parameters.

What is a path-dependent type?

What are they subtypes of?

A type sensitive to the path used to access it.

They are subtypes of some more general path-independent type.

For example, `myOuter.Inner` is a path-dependent subtype of `Outer#Inner`.

What is a self type?

What is its syntax?

What is its purpose?

The assumed type of this, to be used within a trait or class.

```
trait MyTrait { this: SomeType...
```

It can be used to limit the concrete classes that mix in the trait.

What can be abstract in Scala?

Which need be declared abstract?

Classes, methods, fields, types.

Only classes are declared abstract. Methods of traits can be too, in the abstract override case.

Traits are by definition abstract.

What are the types of enum values?

The path-dependent type.

`MyEnumSingleton.Value`

Value is an inner class of Enumeration and has a similarly named method to create new instances.

How do you make an enumeration?

You extend `Enumeration` with a singleton and create `vals`, assigning each the result of the `Value` method.

```
objectColor extends Enumeration {  
  val Red, Blue, Green = Value  
}
```

What's the limitation of pre-initialized fields?

This is exactly like what?

Initializers can't refer to the object to be constructed, because initializers run before superclass constructor.

Hence "this" refers to object *containing* the class or object being constructed.

This is exactly like class constructor args.

Give the syntax of pre-initialized fields.

Put fields in curly braces before superclass constructor call. Use 'with' for classes too!

```
new {  
    val i = 0  
} with MyType  
object O extends {  
    val i = 0  
} with MyType  
class C extends {  
    val i = 0  
} with MyType
```

Give the initialization order for class parameters and abstract fields.

Class parameters are evaluated before they are passed to the class constructor.

An implementing val definition in a subclass is evaluated only after the superclass has been initialized.

This matters when vals are not simple literals.

What do pre-initialized fields do?

When is this frequently useful?

They allow initialization of a field *before* a superclass is called.

This is typically useful when "instantiating" a trait (via an anonymous class) where the trait's initializers reference its abstract fields.