- 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.

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

`Outer#Inner` unlike Java's `Outer.Inner`.

'.' is reserved for objects in Scala.

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
  }
}
```

- Named values. Create with `Value("name")` instead of Value.
- `retrieving Int: Color.Red.id`
- `retrieving value: Color(I)`

Pre-initialized fields, lazy `val`s.

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.

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 there is lazy code run.

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.

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

In other words, abstract `val`s constrain their legal implementations.

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.

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

Abstract vals combined with anonymous classes.

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

Only because traits have constructors but no parameters.

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`.

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.

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.

The path-dependent type.

```
MyEnumSingleton.Value
```

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

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

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

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.

Put fields in curlies 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
```

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.

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.