What's the general rule for mix in order?

Traits further to the right take effect first. If that method calls super, it invokes the method to its left, and so on.

What do stackable modifications allow?

What is a consequence?

They allow modification of a class's behavior by one or more traits without defining a new version of the class via overridden methods in the class body.

Hence the modified class may well have no body. The trait does all the modification for you.

Give the mix in syntax.

```
class X extends someClass with Trait 1 with
Trait 2...
```
or
```
class X extends Trait 1
```
or
```
class X extends Trait 1 with Trait 2
```

What is the shortest class definition?

classX

What's the main thing you need to know about linearization sequence?

A class is always linearized before *all* of its superclasses and mixed in traits.

Thus when you write a method that calls super, the method is definitely modifying the behavior of the superclasses and mixed in traits, not the other way around.

Give two ways traits differ from classes.

- They have class parameters.
- Super calls are dynamically instead of statically bound.

Trait vs. mixin

Once a trait is mixed into a class, you can alternatively call it a mixin.

Describe "super" method calls in abstract methods of classes and traits.

In classes this is not allowed because the call will fail at runtime.

It is allowed in traits with abstract override modifier. Super is dynamically bound so that it will work if the mix-in happens after the class or other trait provides the implementation.

Unlike Java interfaces, what can Scala traits do?

- Extend a superclass.
- Have method implementations.
- Have state.
- Have a constructor (sans parameters).

What does the `Ordered[T]` trait get you?

<, >, <=, >=

You only implement `compare(T)`.

Returns
`x < 0 iff this < that
x = 0 iff this == that
x > 0 iff this > that`

How do trait superclasses affect client code?

`trait X extends Y` means X can only be mixed into a Y.

What's a major problem in implementing stackable modifications in a multiple inheritance language?

With multiple inheritance, the method super calls can be determined statically. With traits, linearization determines the calls.

Linearizing classes with traditional multiple inheritance would break with normal static interpretation of super that is otherwise used. Scala can use one strategy for classes and one for traits.

Why can't `Ordered[T]` override `equals()` for you?

Type erasure!

`equals` has to check the type of the argument to see it it's a T, which is not possible?

This check must be performed because `compare(T)` takes a narrower type than `equals`, in general.

What are some common uses of traits?

They enrich thin interfaces without burdening the client, since most methods can come fully implemented.

They can define stackable modifications.

How should you choose between traits and classes?

Traits should be the default choice, as they keep open more options.

```
abstract class                          | traits
----------------------------------------------------------------------------
inherit from Java                       | reusable in multiple parts of the
□    □                 | hierarchy
distribute your code in compiled    |
form without forcing recompilation | usable from Java if only made of
on every add/remove of member□   | abstract methods
   □□      □                |
efficiency□                 |
```