Mutable vs Immutable

Objects have state, which is the data stored in instance fields.

State can change after an object is created, either intentionally or unintentionally.

When state remains constant throughout the lifetime of the object, and code is prevented from changing the state, this object is called an immutable object.

An immutable object is an object whose internal state remains constant.

A mutable object is an object whose internal state does not remain constant.

Which is better?

Well, like anything else, that depends.

Immutable Objects - Advantages

Working with immutable objects has some advantages.

An immutable object isn't subject to unwanted, unplanned and unintended modifications, known as side-effects.

An immutable class is inherently thread-safe, because no threads at all can change it, once it's been constructed.

This allows us to use more efficient collections and operations, which don't have to manage synchronization of access to this object.

These are two of the most important advantages.

Immutable Objects - Disadvantages

Working with immutable objects has some disadvantages.

An immutable object can’t be modified after it's been created.

This means that when a new value is needed, you're probably going to need to make a copy of the object with the new value.

You'll remember the discussion comparing String vs. StringBuilder.

If you're constantly needing to alter text values, it's more efficient to use a mutable object like a StringBuilder instance, then generating tons of new String objects.

Classes must be designed to produce Immutable Objects

I'll be talking about immutable class design coming up.

It's important to understand that POJOs and Java Beans, which many code generation tools create, are not by design, immutable, and therefore this architecture may not be the preferred design for your class.

This all sounds rather simple, yet there are many topics related to it.

One of the most useful tools in our arsenal to build immutable classes, is the final access modifier.

I'll be revisiting this modifier in much more detail in the next video, so let's get going.