The Object instance monitor

Every object instance in Java has a built-in *intrinsic* lock, also known as a monitor lock.

A thread acquires a lock by executing a synchronized method on the instance, or by using the instance as the parameter to a synchronized statement.

A thread releases a lock when it exits from a synchronized block or method, even if it throws an exception.

Only one thread at a time can acquire this lock, which prevents all other threads from accessing the instance's state, until the lock is released.

All other threads, which want access to the instance's state through synchronized code, will block, and wait, until they can acquire a lock.

The synchronized statement can be applied to a more granular code block

The synchronized statement is usually a better option in most circumstances, since it limits the scope of synchronization, to the critical section of code.

In other words, it gives you much more granular control, over when you want other threads to block.

The synchronized statement can be applied to a more granular object in state

The synchronized block can use a different object, on which to acquire its lock.

This means that code, accessing this bank account instance, wouldn't have to block entirely.

Reentrant Synchronization

I said that once a thread acquires a lock, all other threads will block, which also require that lock.

Because these method calls are executed from the **same thread**, any nested calls which try to acquire the lock, won't block, because the current thread already has it.

This feature is called Reentrant Synchronization.

Without this, threads could block indefinitely.

This concept is built into the Java language, and it's based on the monitor mechanism.