Applying Concurrency and Multi-threading to Common Java Patterns

UNDERSTANDING CONCURRENCY, THREADING, AND SYNCHRONIZATION

José Paumard
PHD, JAVA CHAMPION, JAVA ROCK STAR

@JosePaumard http://blog.paumard.org/

Concurrency: “the art of doing several things at the same time”

What does correct code mean in the concurrent world

How to improve your code by leveraging multi-core CPUs

Writing code, implementing patterns

Race condition, synchronization, volatility

Visibility, false sharing, happens-before

This is a Java course

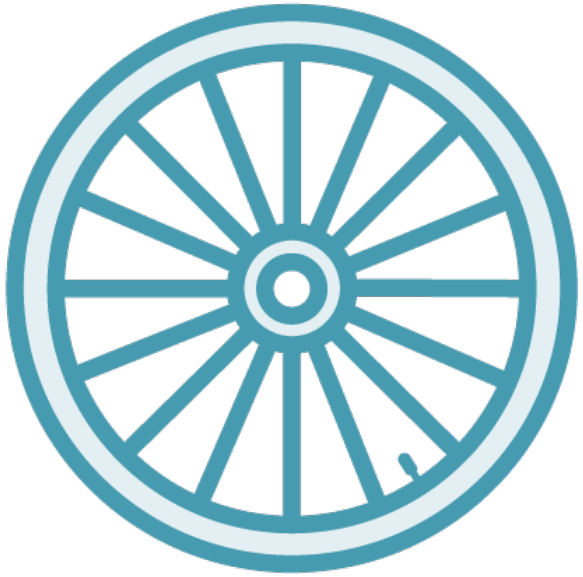
Fair knowledge of the language and its main API

No prior knowledge about concurrency

Agenda

- Understanding concurrency, threading and synchronization
- Implementing the producer / consumer pattern using wait / notify
- Ordering reads and writes operations on a multicore CPU
- Implementing a thread safe singleton on a multicore CPU

What Is a Thread?



A thread is defined at the Operating System level

A thread is a set of instructions

An application can be composed of several threads

Different threads can be executed “at the same time”

The Java Virtual Machine works with several threads (GC, JIT, ...)

What Does “At the Same Time” Mean?

Writing a text document

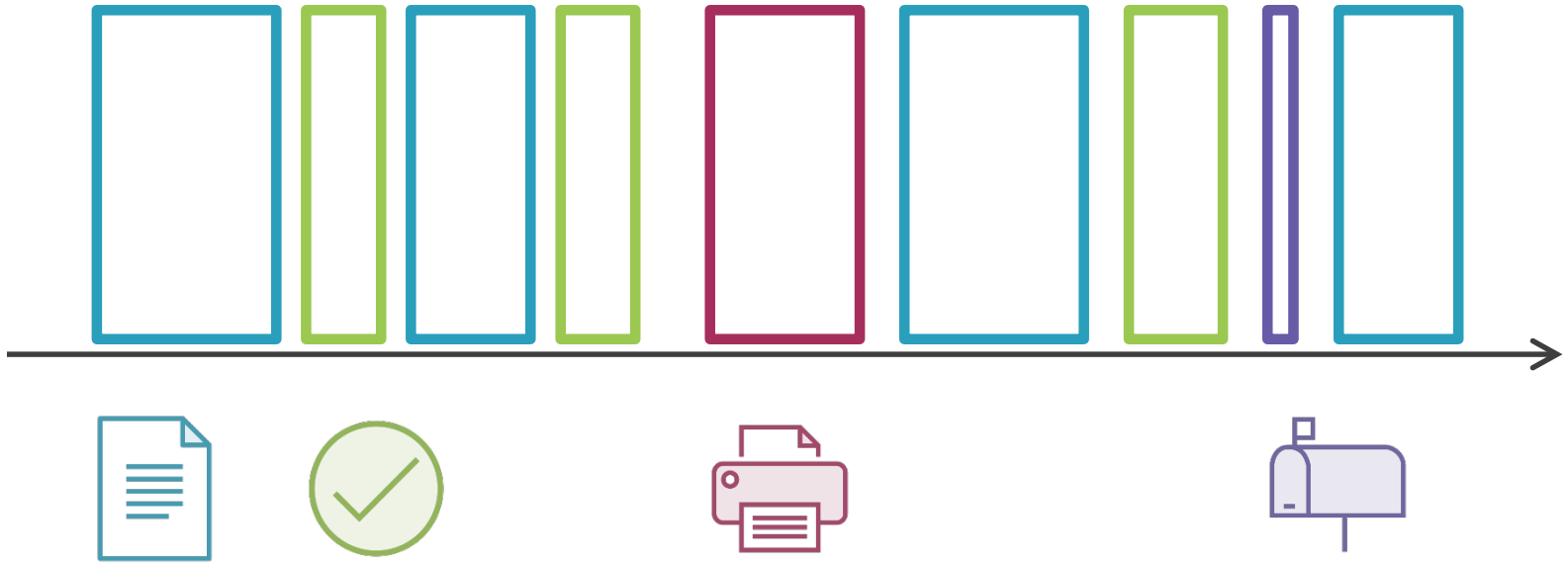
Running the spell check

Printing elements of the document

Receiving emails

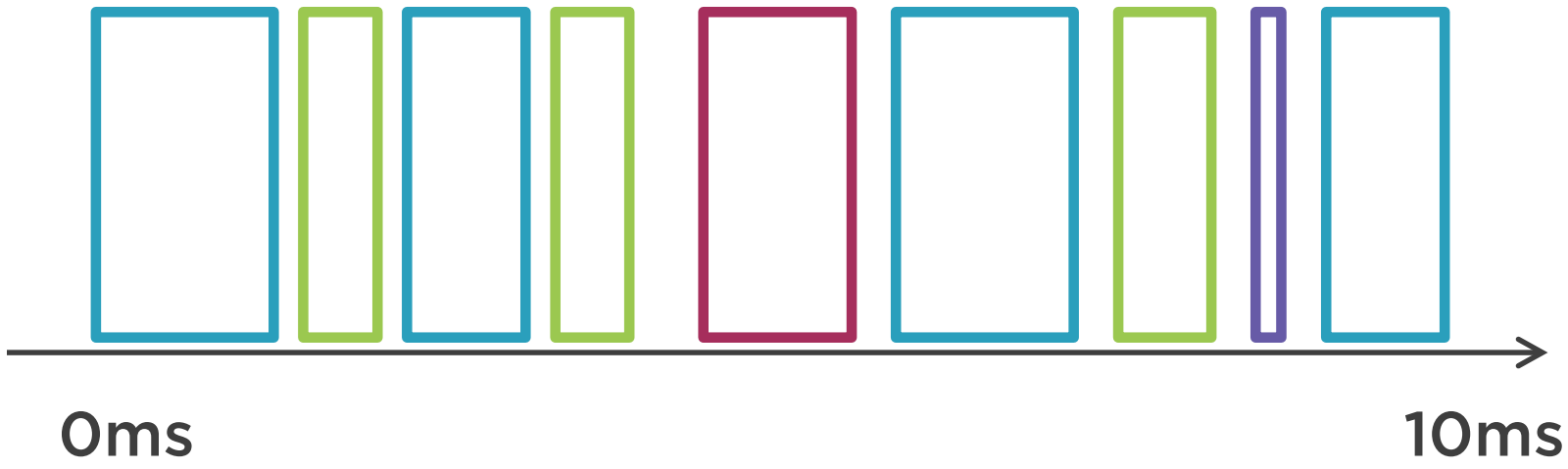
What is happening at the CPU level?

1st Case: CPU with Only One Core

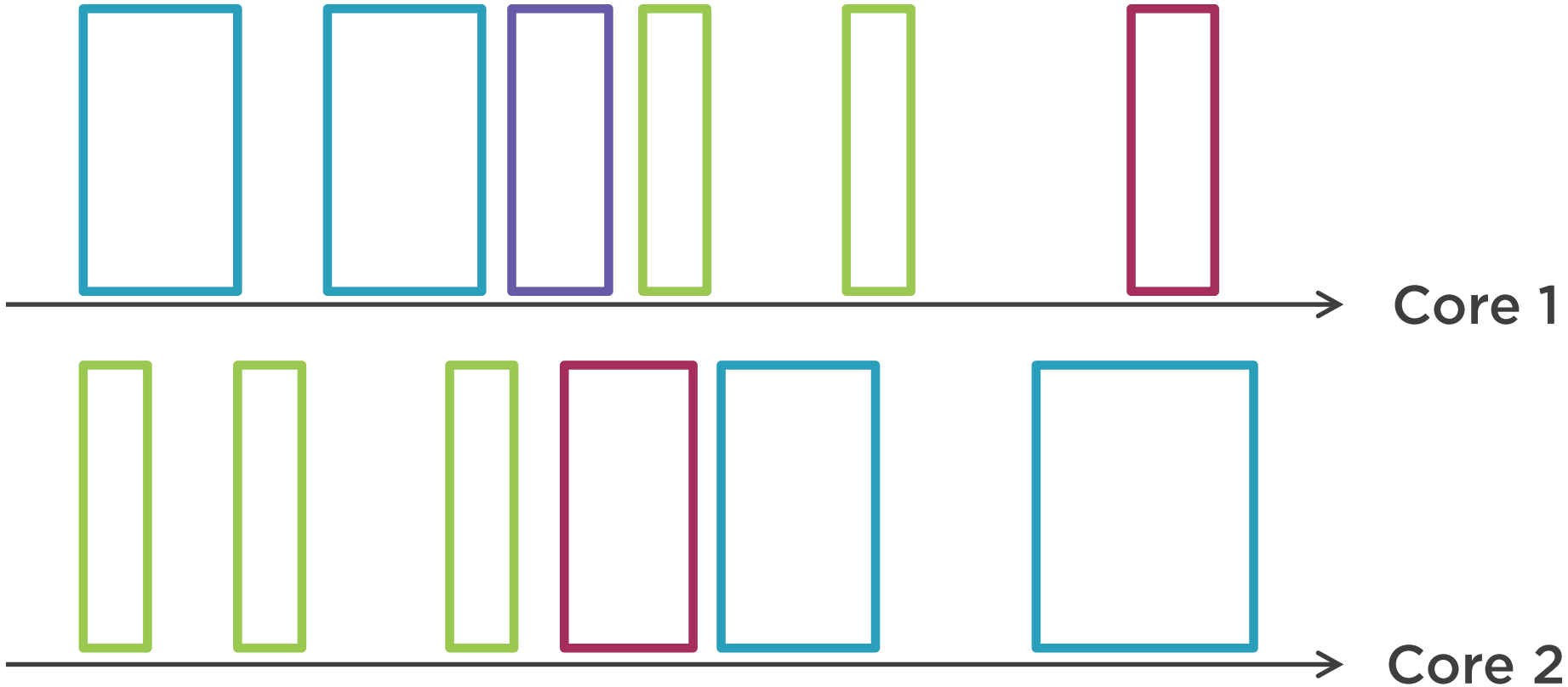


Why do we have the feeling that everything is happening at the same time?

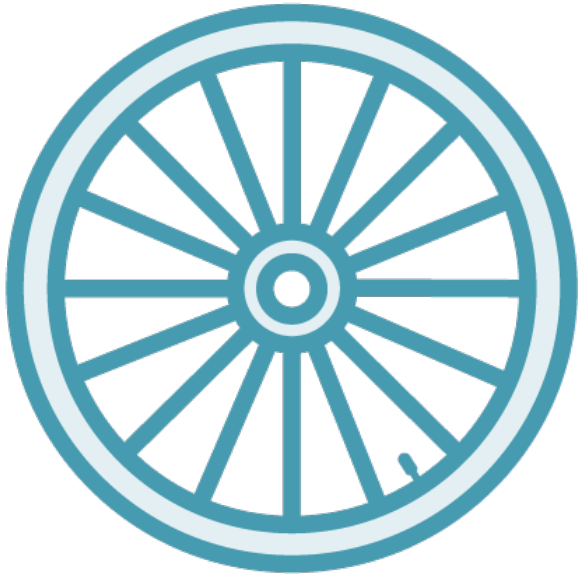
Because Things Are Happening Fast!



2nd Case: CPU with Multiple Cores



Only on a multicore CPU are things happening “at the same time”



Who is responsible for the CPU sharing?

A special element called a scheduler

There are three reasons for the scheduler to pause a thread:

The CPU should be shared equally among threads

The thread is waiting for some more data

The thread is waiting for another thread to do something

Race Condition

Race Condition Definition

Accessing data concurrently may lead to issues!

It means that two different threads
are trying to read and write
the same variable at the same time

This is called a race condition

“same time” does not mean the same thing
on a single core and on a multi core CPU

Example: the Singleton pattern

The Singleton Pattern

```
public class Singleton {


    private static Singleton instance ;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton() ;
        }
        return instance ;
    }
}
```

What is happening if two threads are calling getInstance()?

Thread T_1	Thread T_2
Checks if instance is null?	Waiting
The answer is yes	
Enters the if block	
The thread scheduler pauses T_1	

Thread T_1	Thread T_2
Checks if instance is null?	Waiting
The answer is yes	
Enters the if block	
The thread scheduler pauses T_1	
	Checks if instance is null?
	The answer is yes
	Enters if the block
	Creates an instance of Singleton
	The thread scheduler pauses T_2

Thread T_1	Thread T_2
Checks if instance is null?	Waiting
The answer is yes	
Enters the if block	
The thread scheduler pauses T_1	
	Checks if instance is null?
	The answer is yes
	Enters if the block
	Creates an instance of Singleton
	The thread scheduler pauses T_2
Create an instance of Singleton	

How to prevent that?

The answer is: synchronization

Synchronization

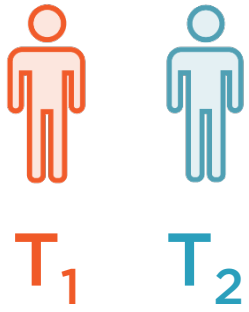
Prevents a block of code to be executed by more than one thread at the same time

The Singleton Pattern

```
public class Singleton {


    private static Singleton instance ;

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton() ;
        }
         return instance ;
    }
}
```

Synchronization

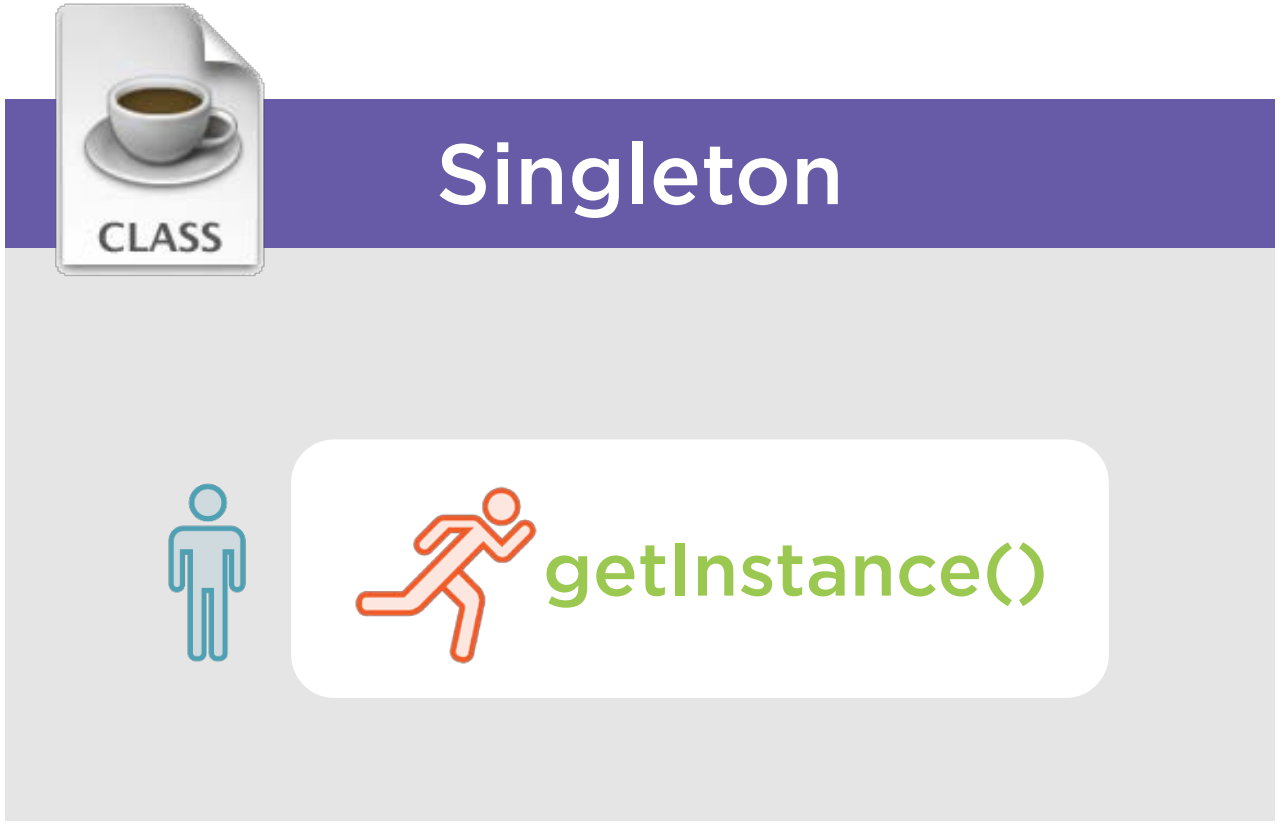
How Does Synchronization Work?



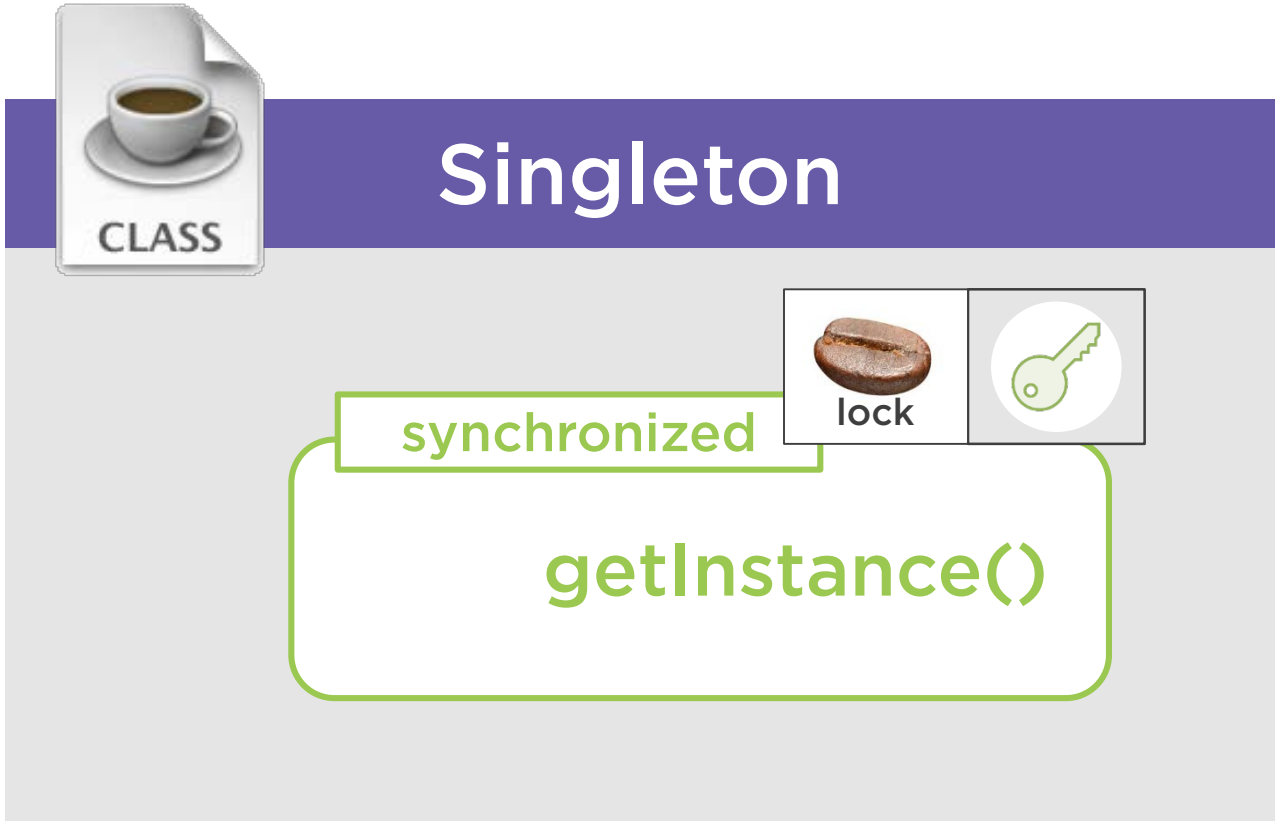
Singleton

getInstance()

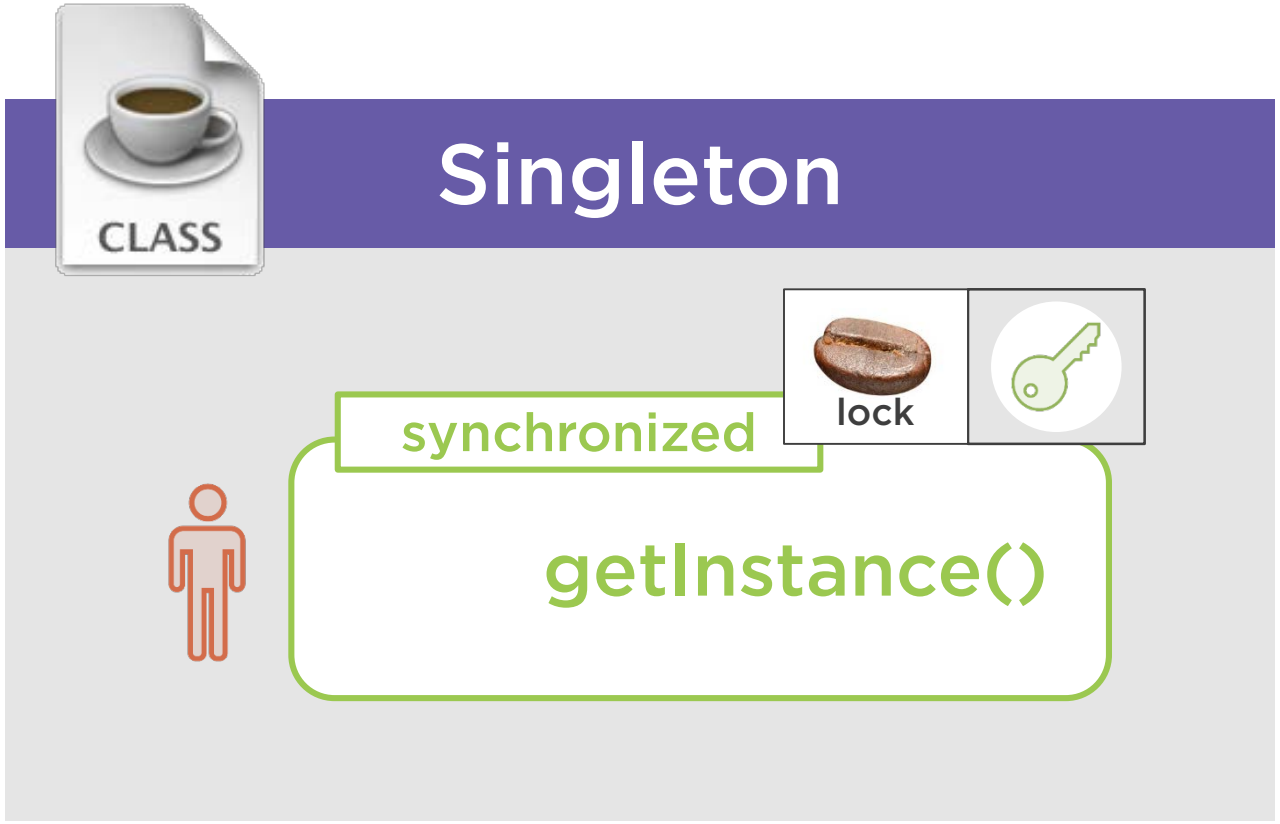
How Does Synchronization Work?



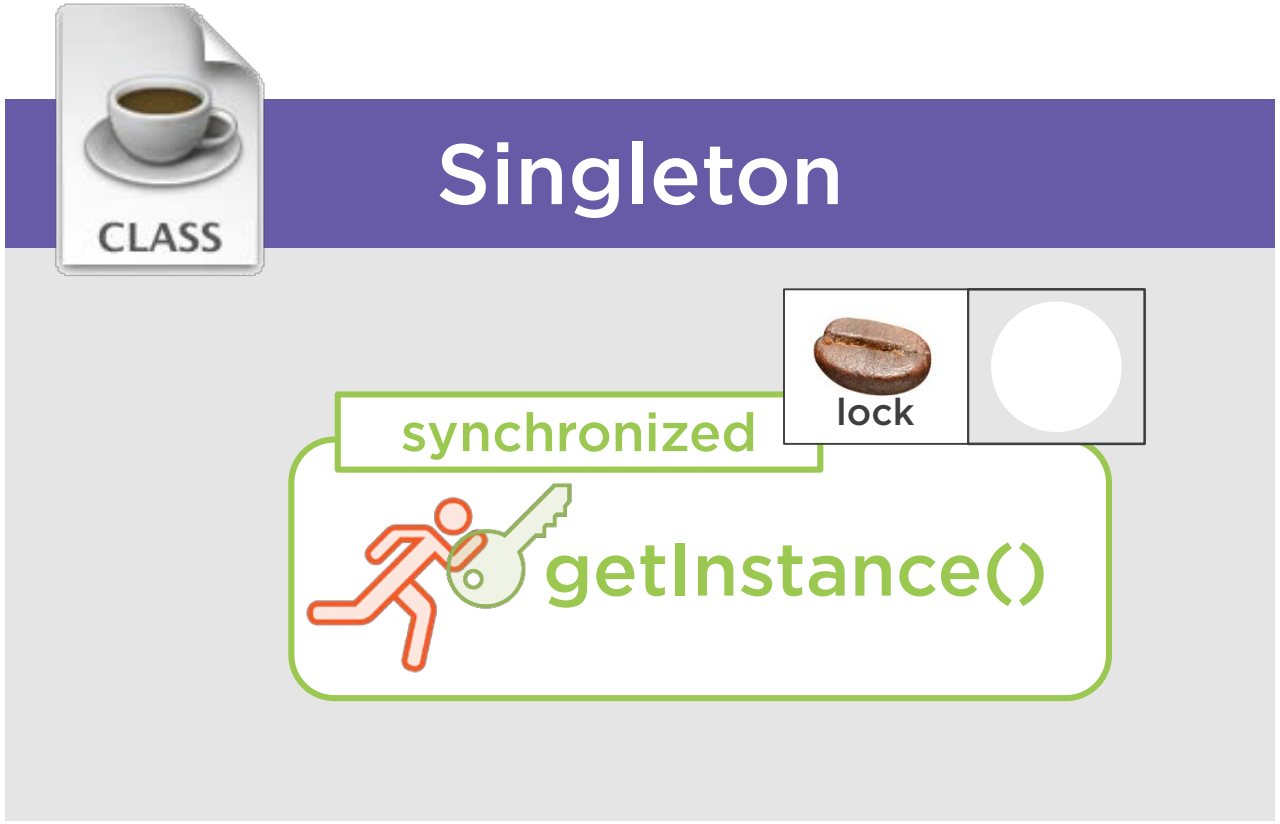
How Does Synchronization Work?



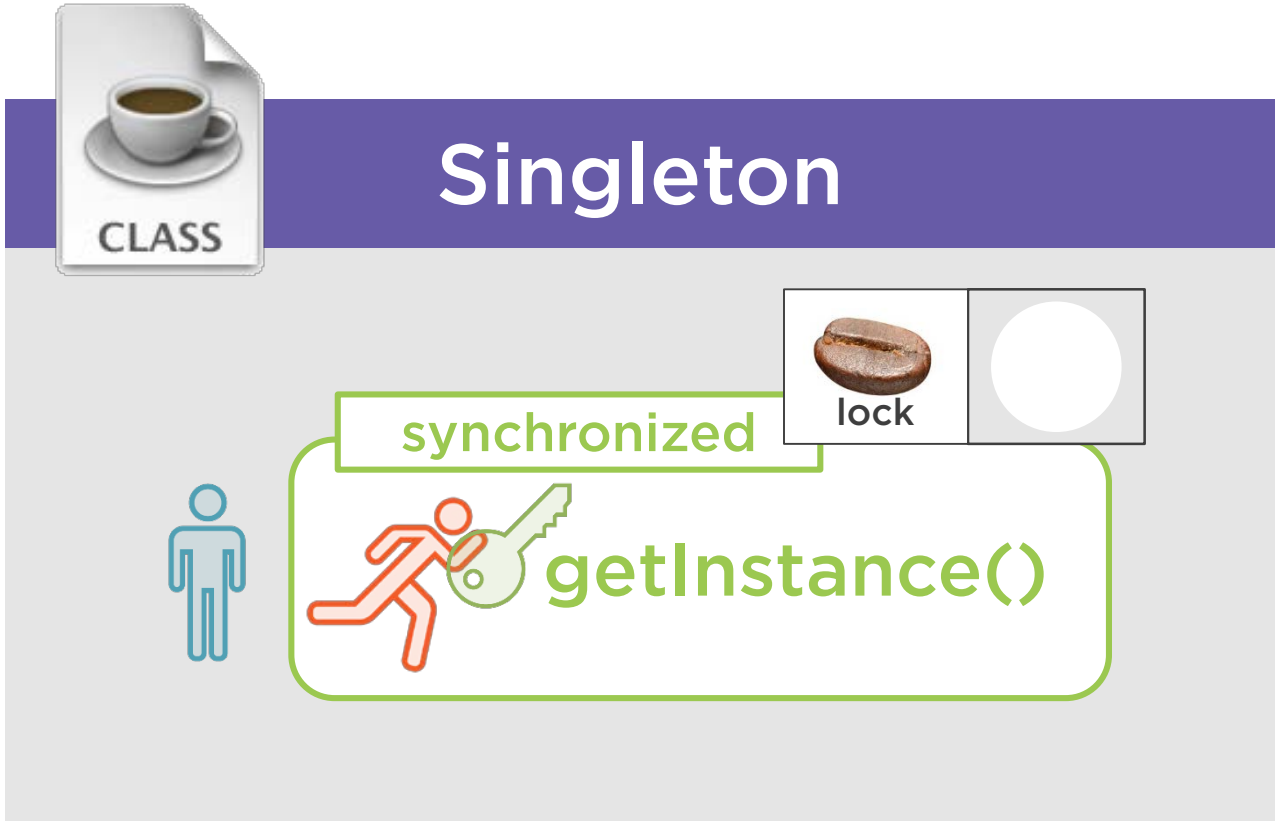
How Does Synchronization Work?



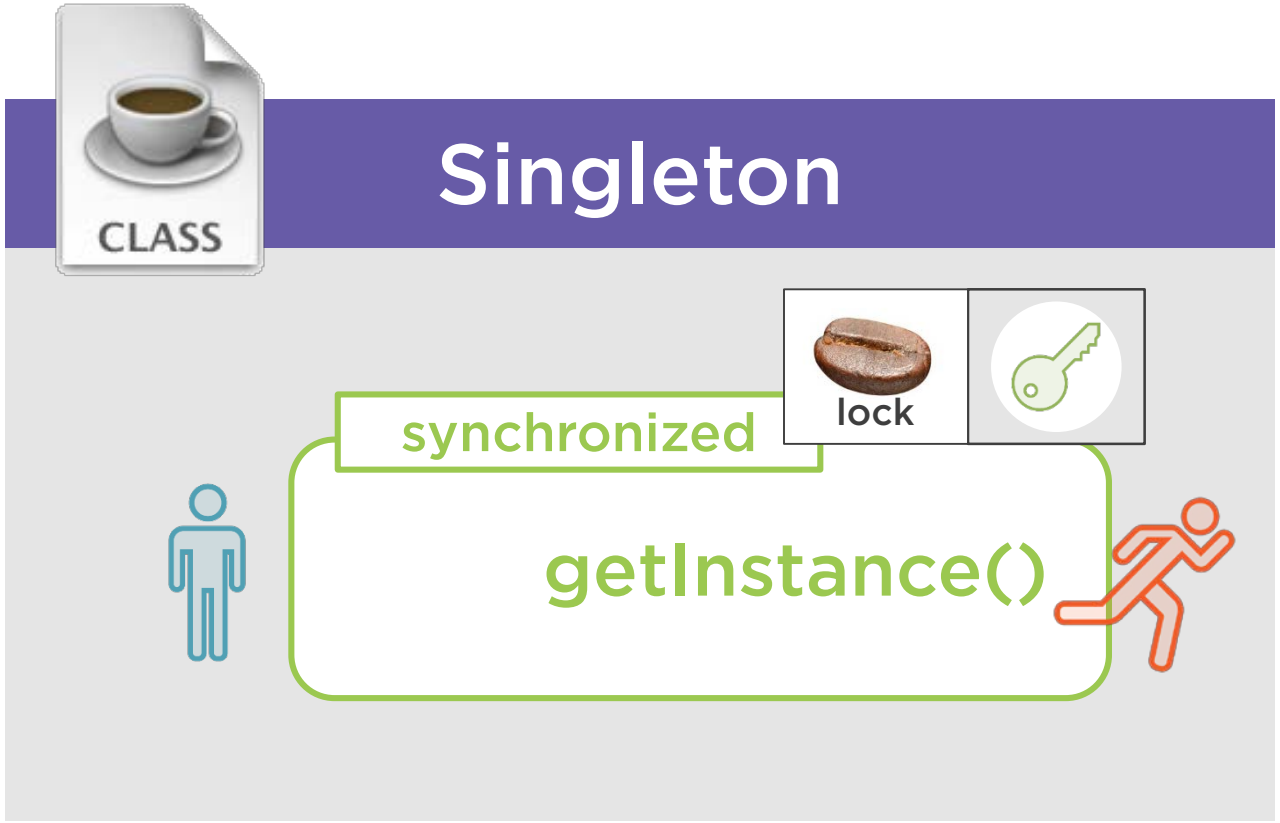
How Does Synchronization Work?



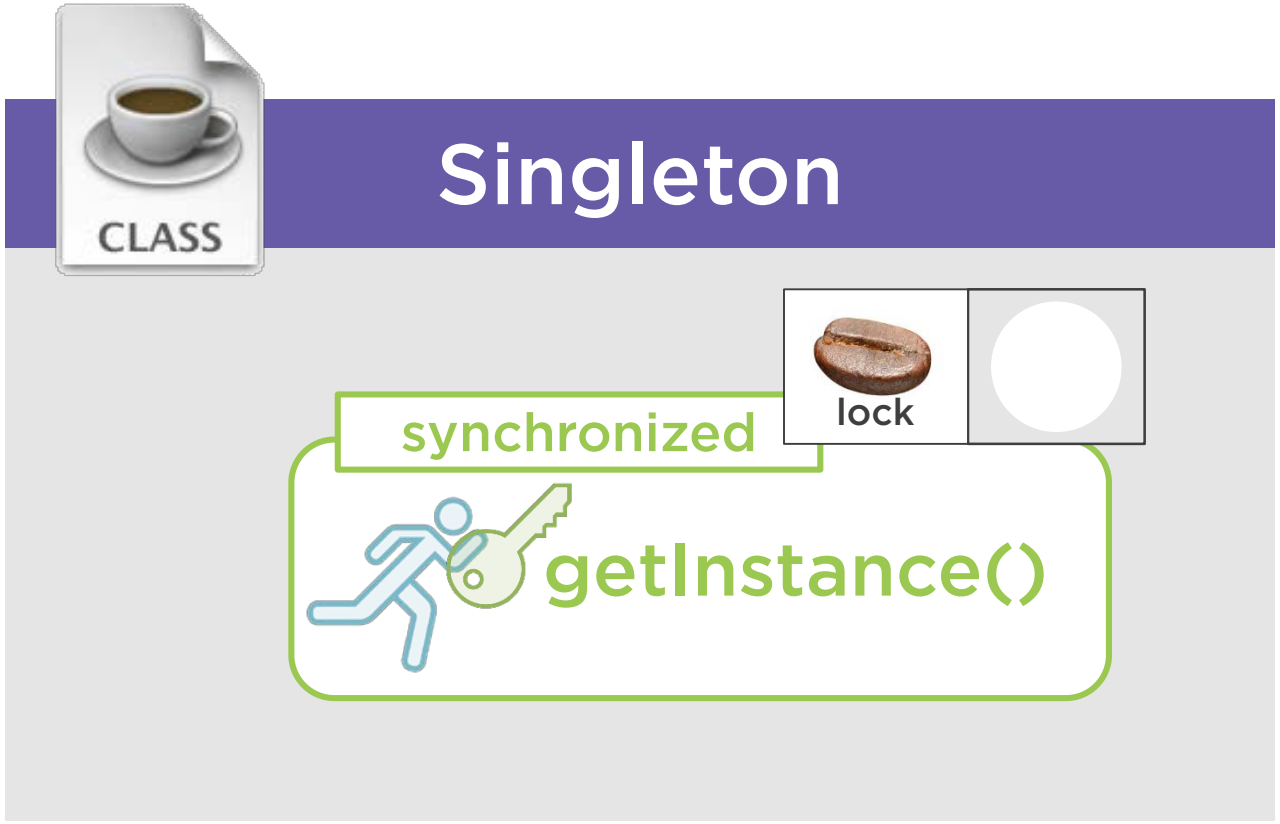
How Does Synchronization Work?



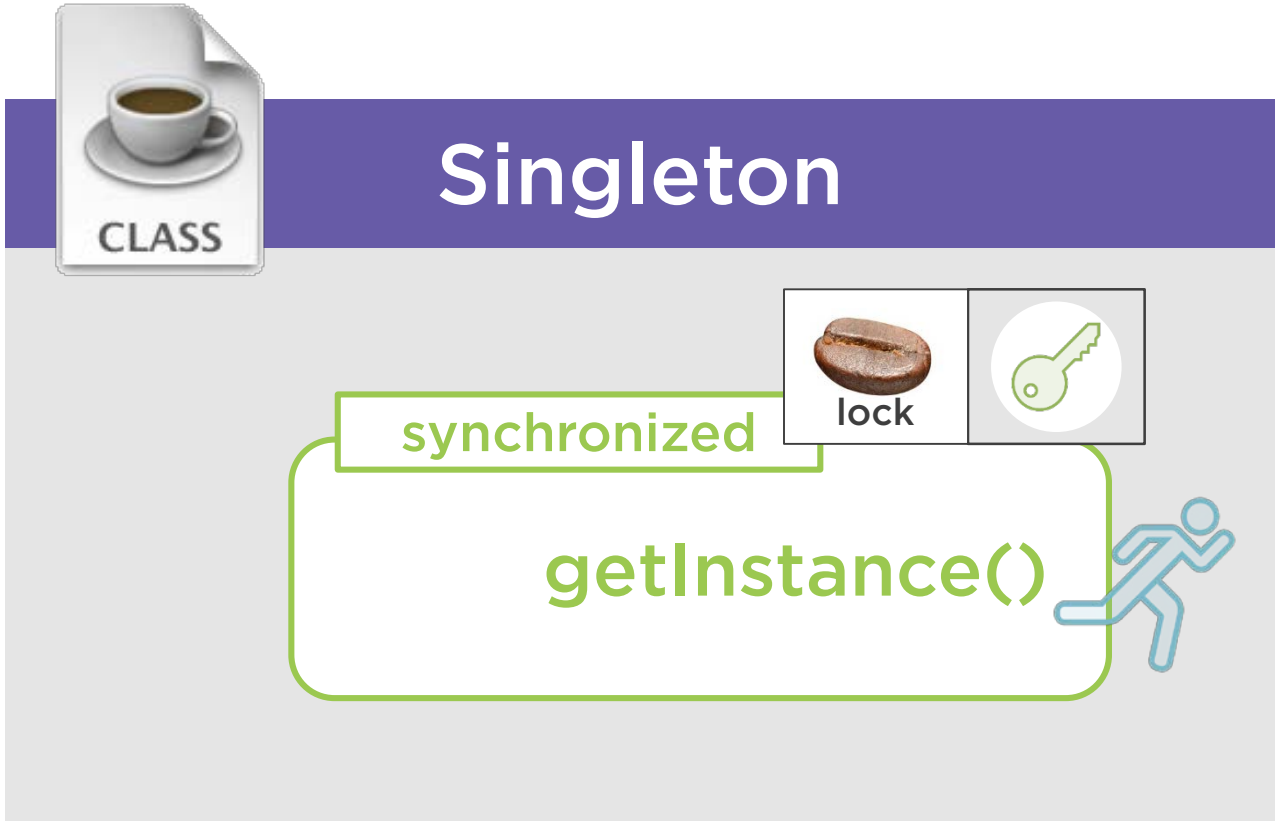
How Does Synchronization Work?

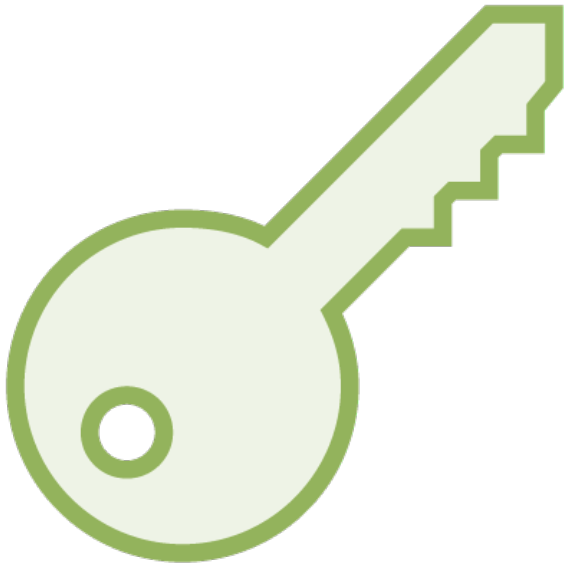


How Does Synchronization Work?



How Does Synchronization Work?





So for synchronization to work, we need a special, technical object that will hold the key

In fact, every Java object can play this role

This key is also called a monitor

How can we designate this object?

```
public static synchronized Singleton getInstance() {
    if (instance == null) {
        instance = new Singleton() ;
    }
    return instance ;
}
```

In this code, the key is the Singleton class itself

A synchronized static method uses the class as a synchronization object

```
public synchronized String getName() {
    return this.name ;
}
```

In this code, the key is the instance of the class

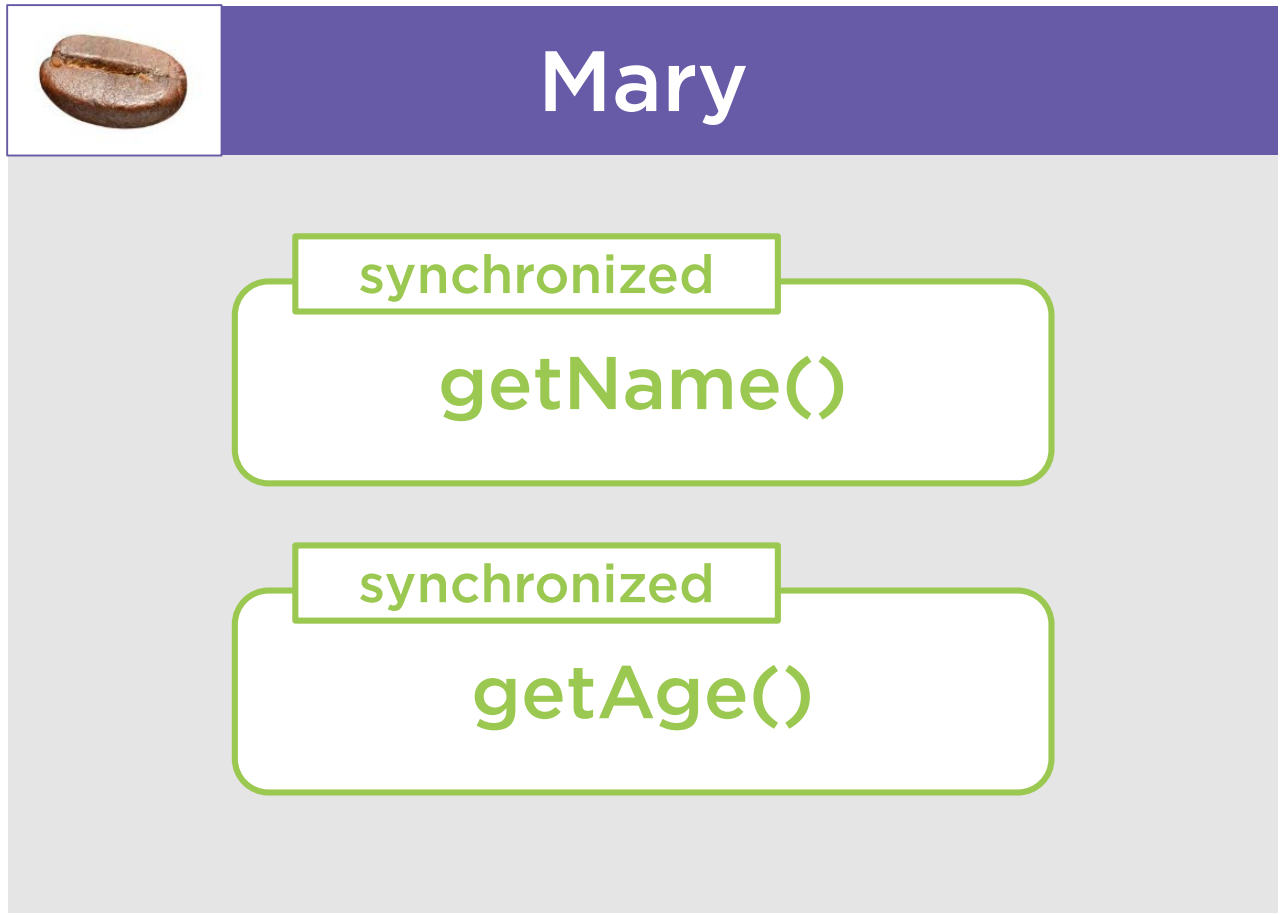
A synchronized non-static method uses the instance as a synchronization object

```
public class Person {
    private final Object key = new Object();

    public String init() {
        synchronized(key) {
            // do some stuff
        }
    }
}
```

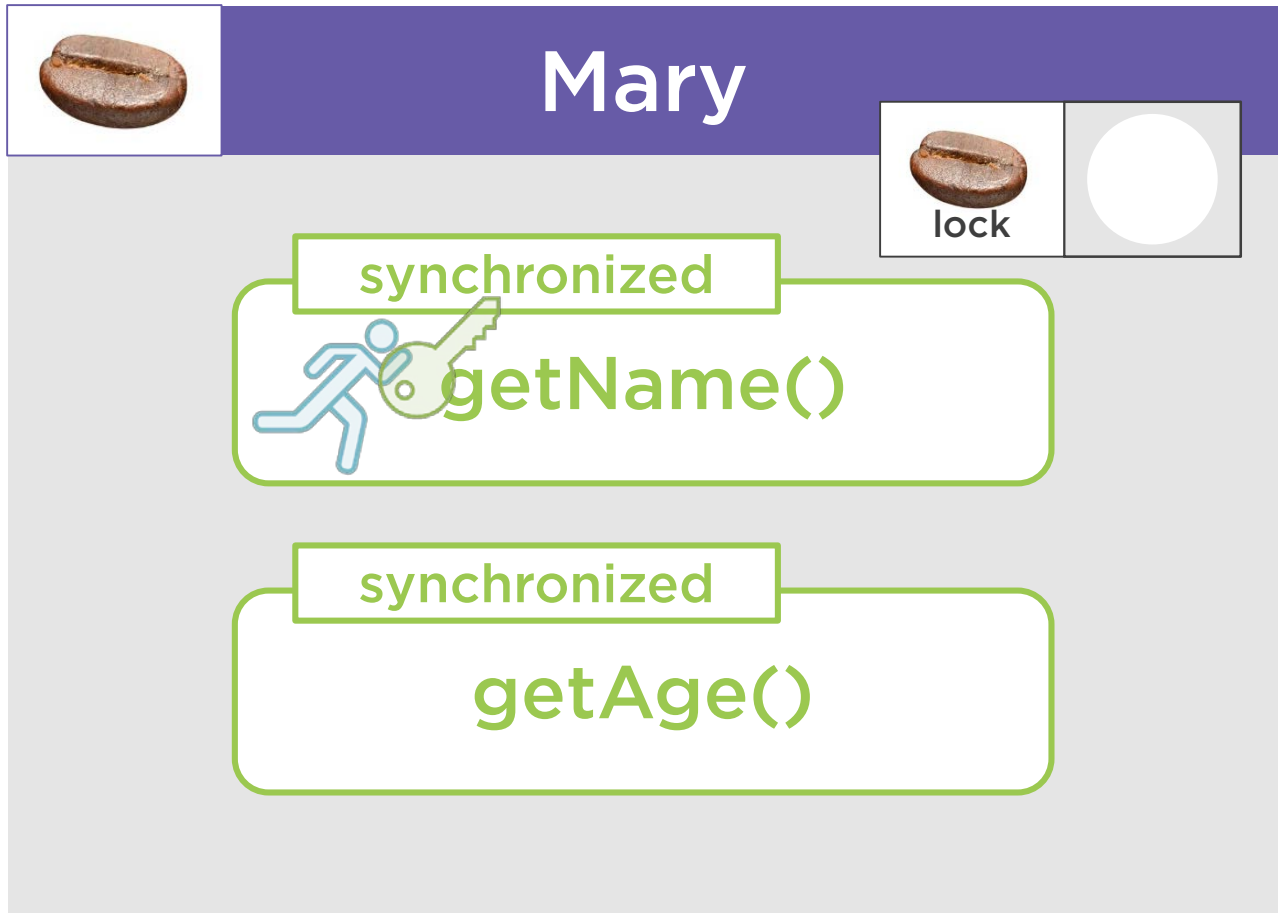
A third possibility is to use a dedicated object to synchronize

It is always a good idea to hide an object used for synchronization

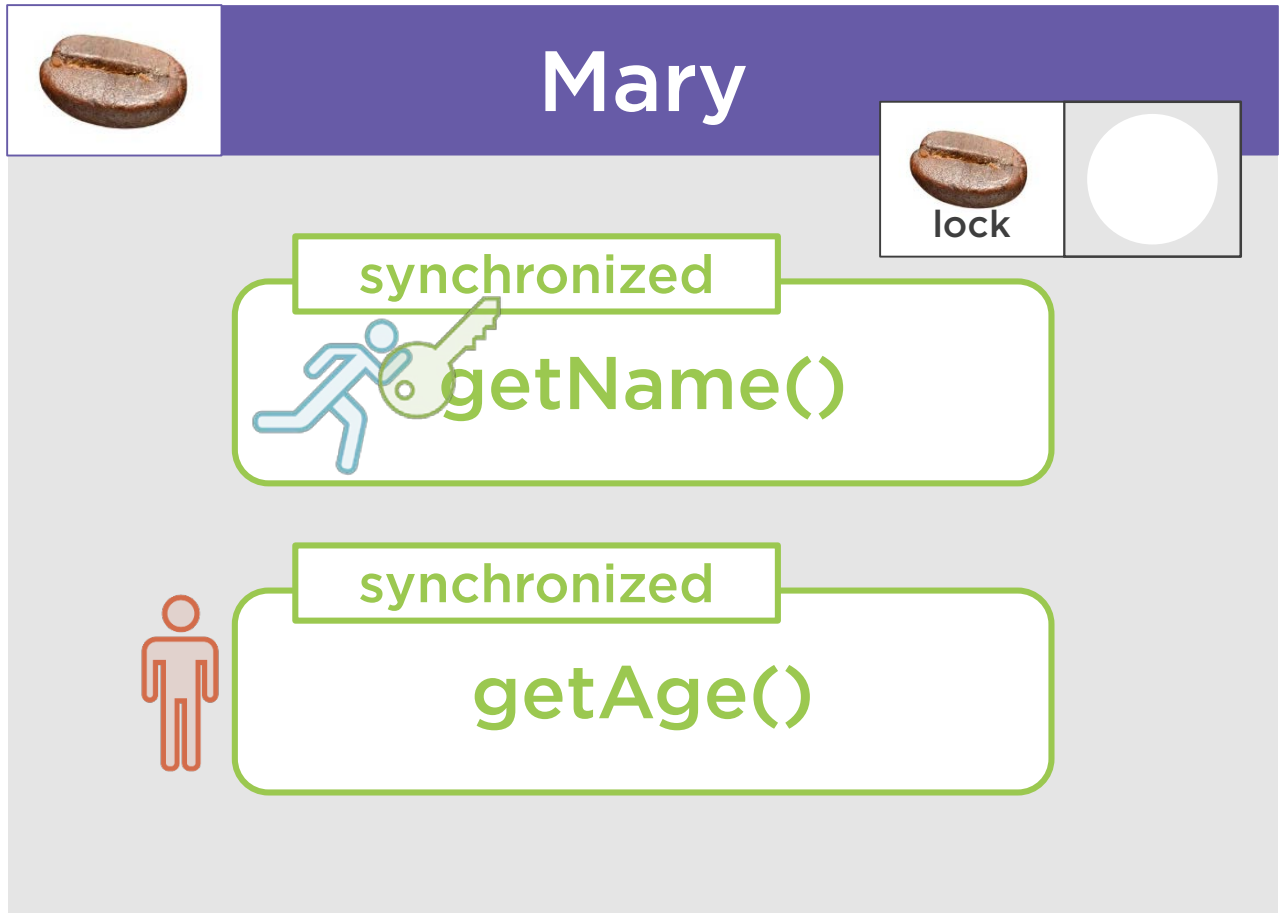
Synchronizing More Than One Method



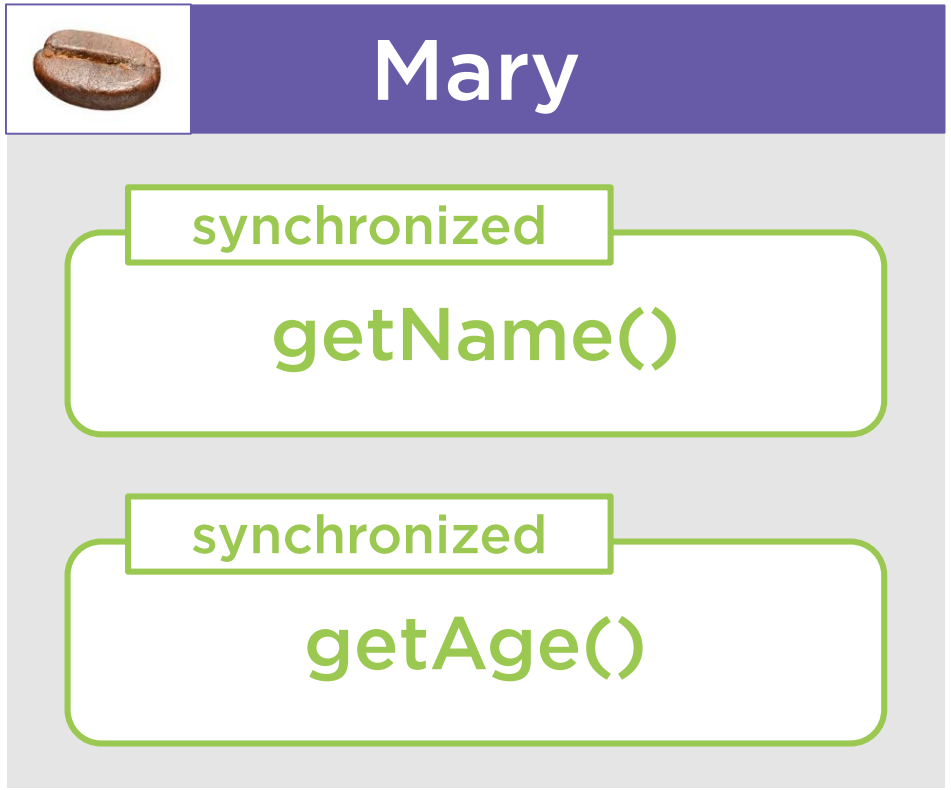
Synchronizing More Than One Method

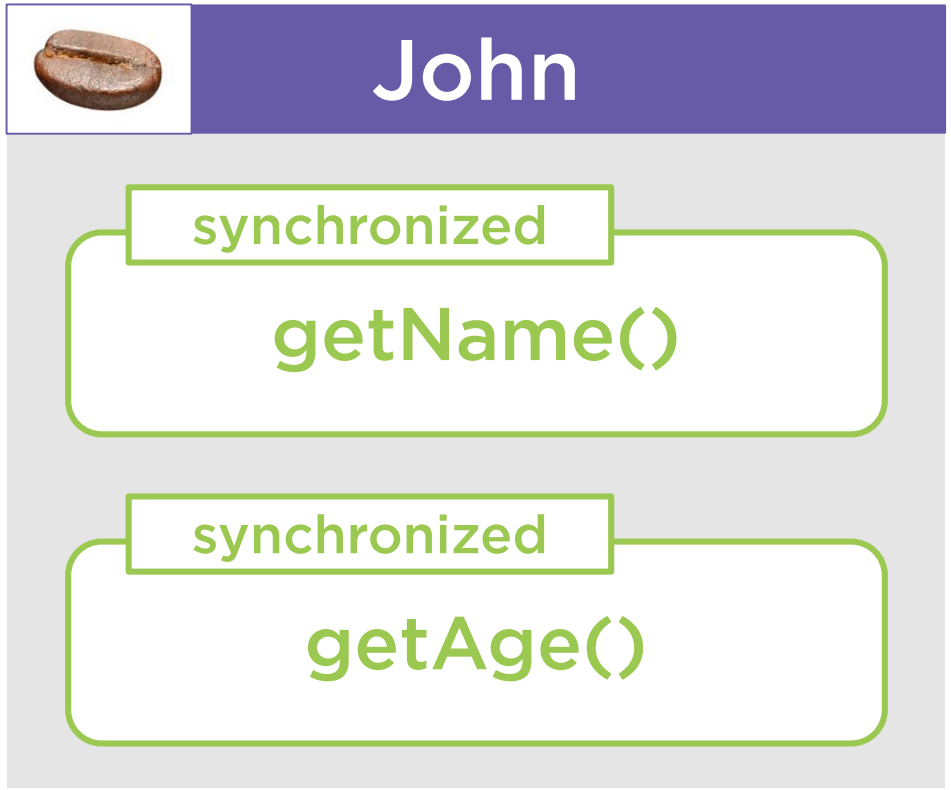


Synchronizing More Than One Method

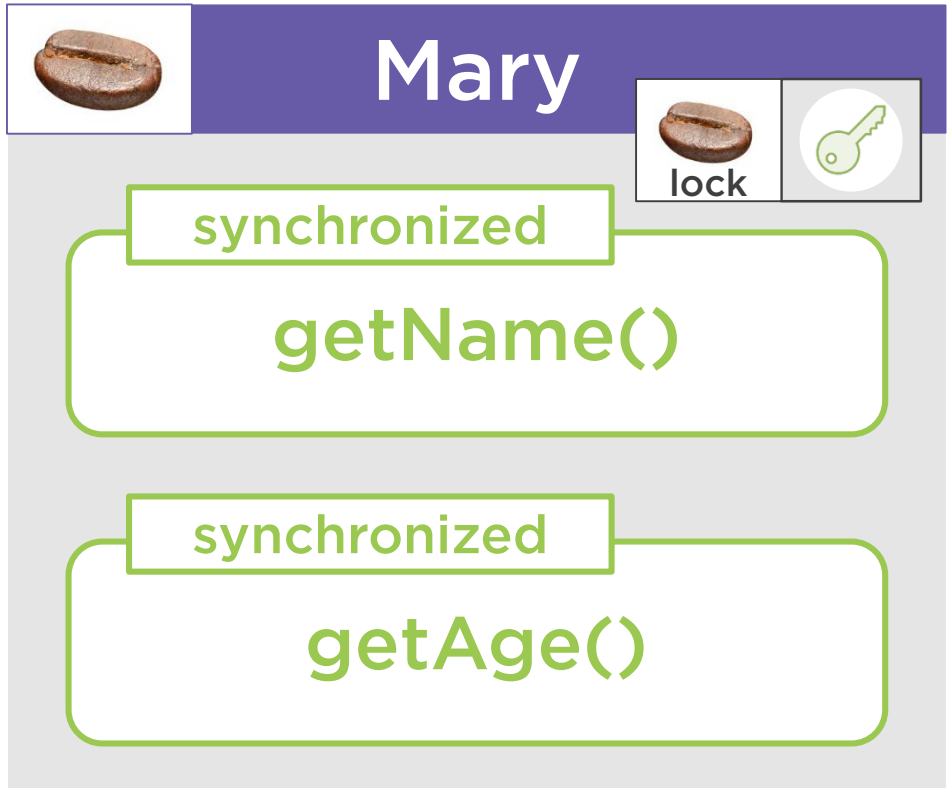


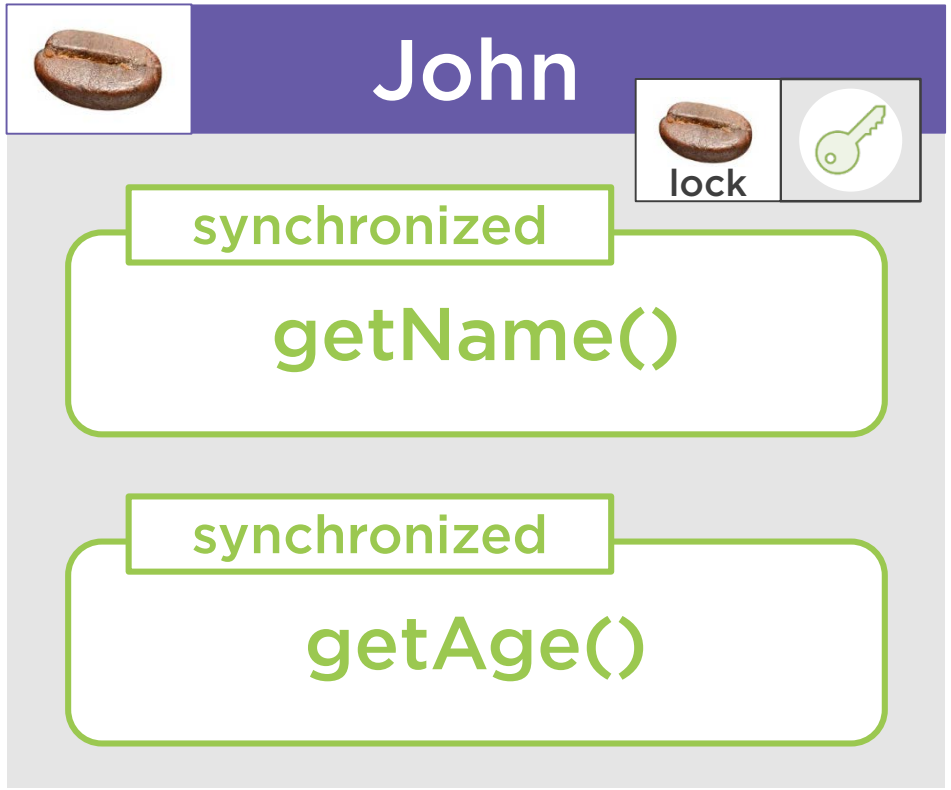
Synchronizing More Than One Method



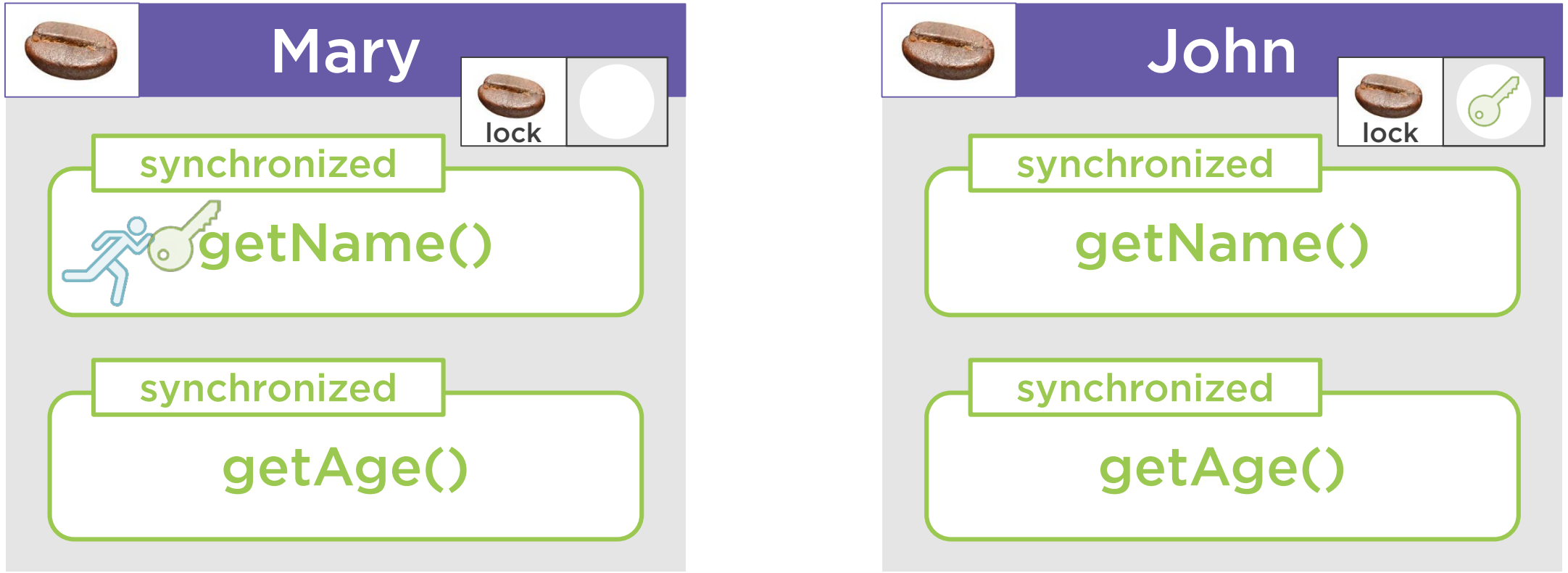


Synchronizing More Than One Method

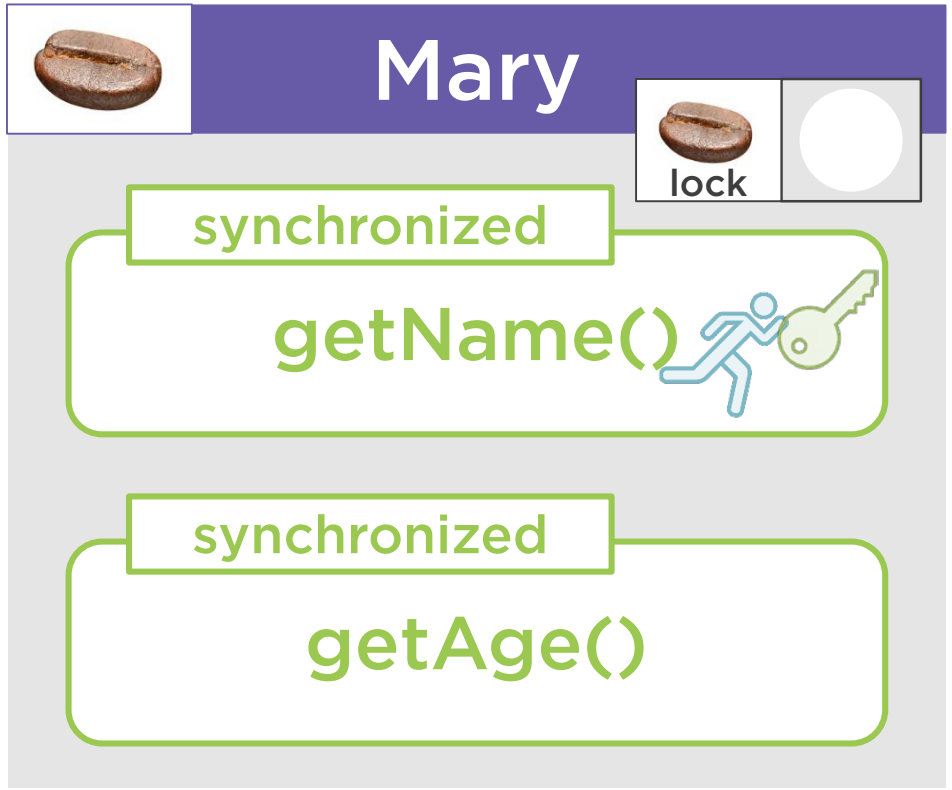


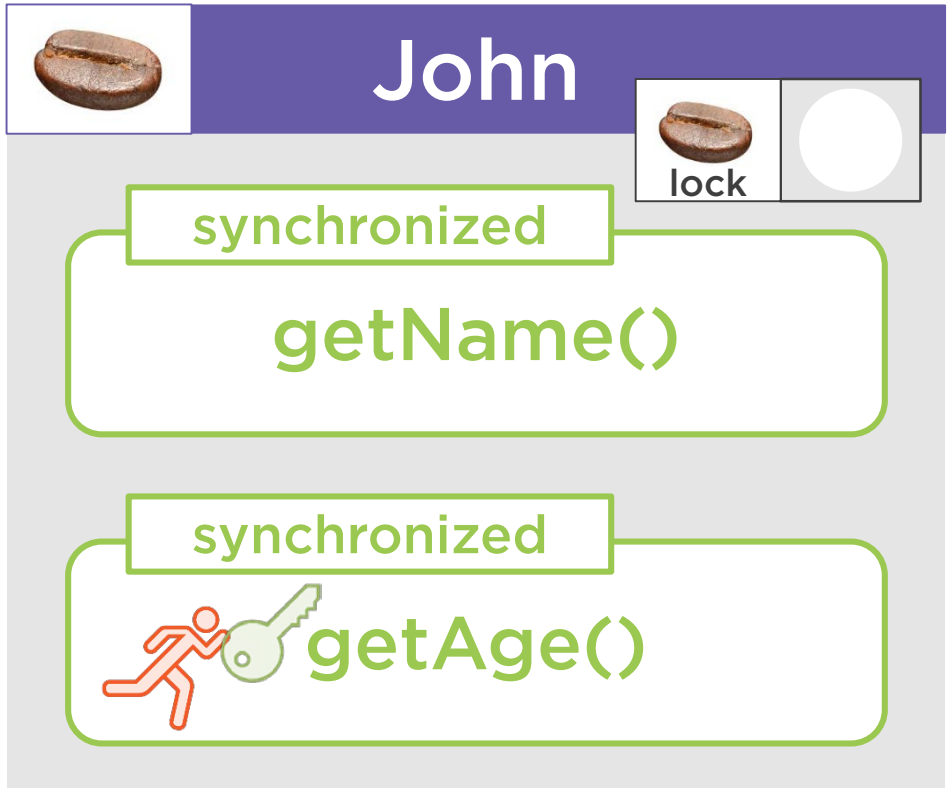


Synchronizing More Than One Method

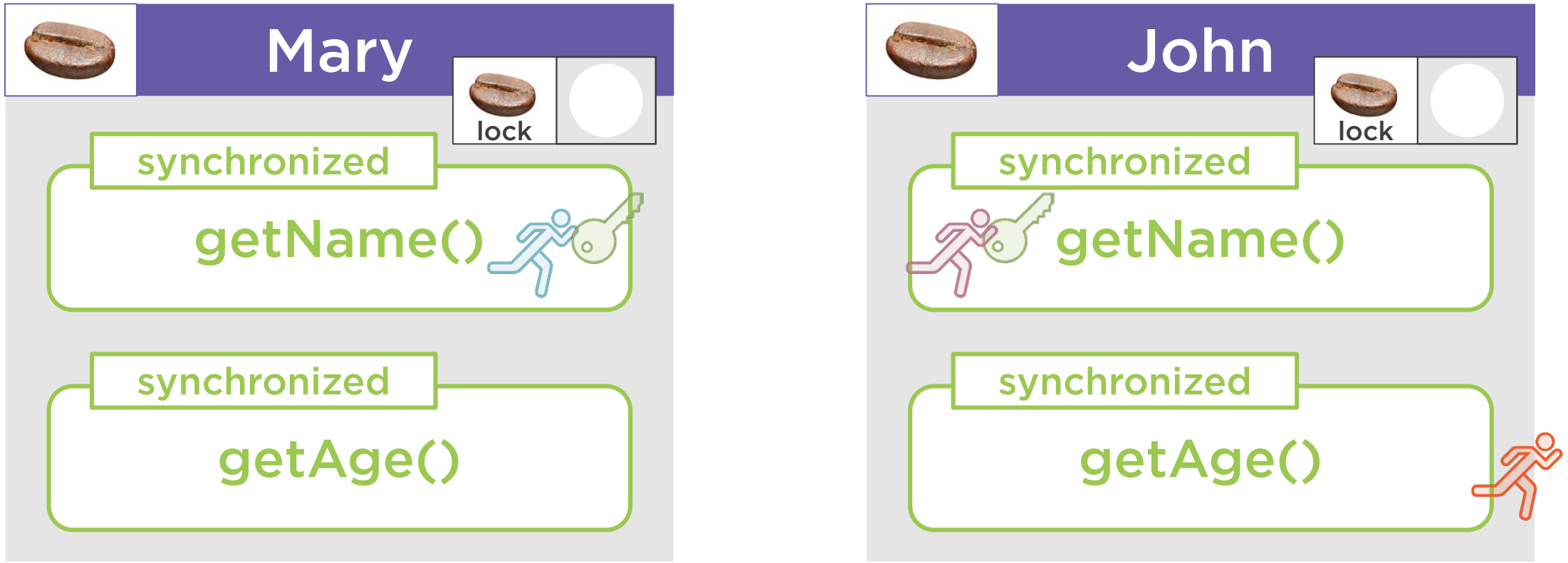


Synchronizing More Than One Method

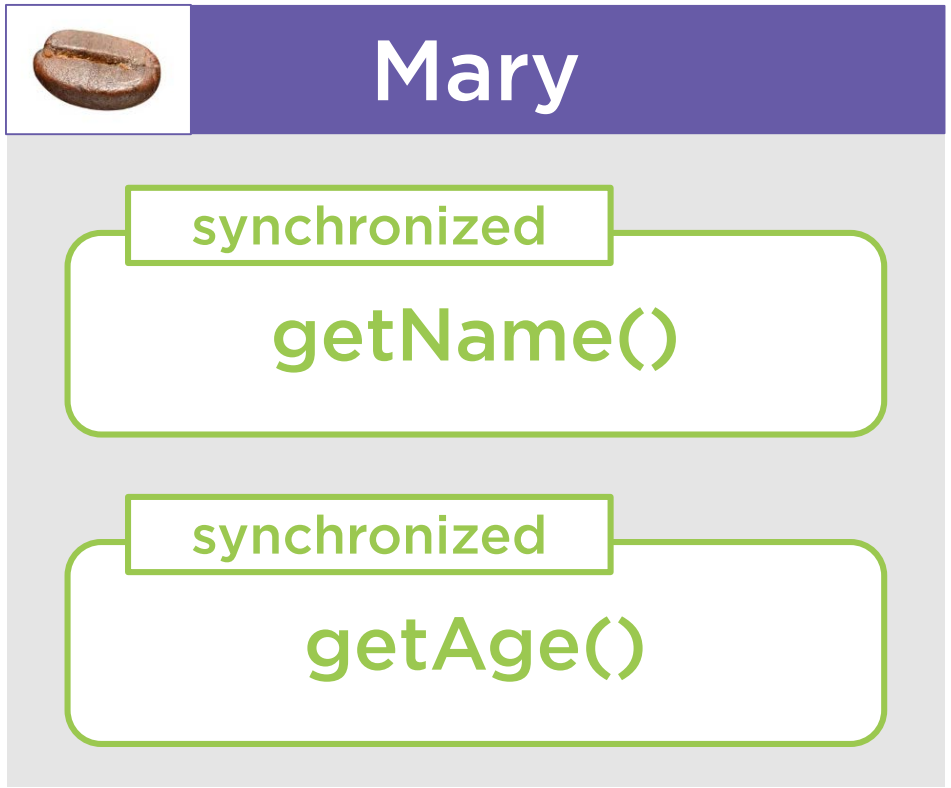


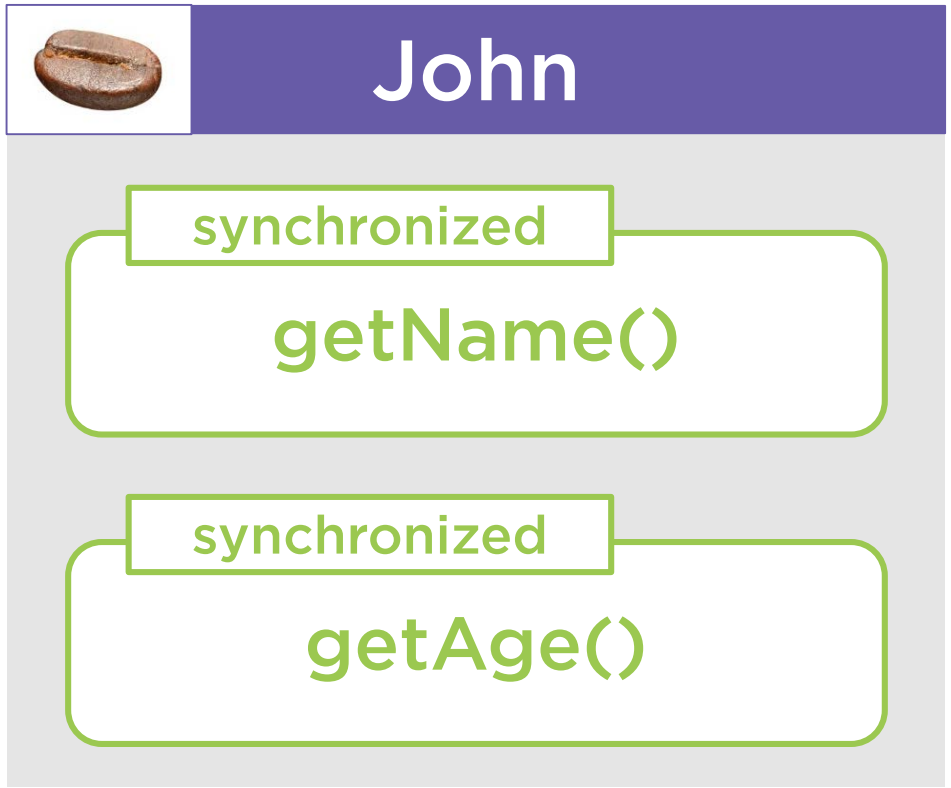


Synchronizing More Than One Method

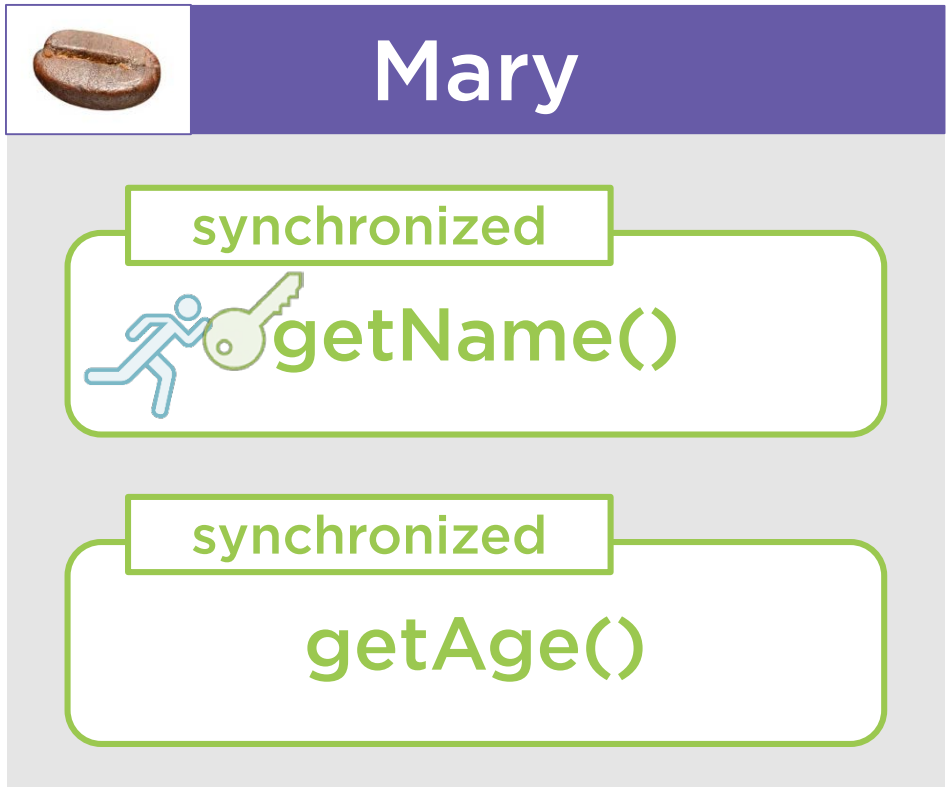


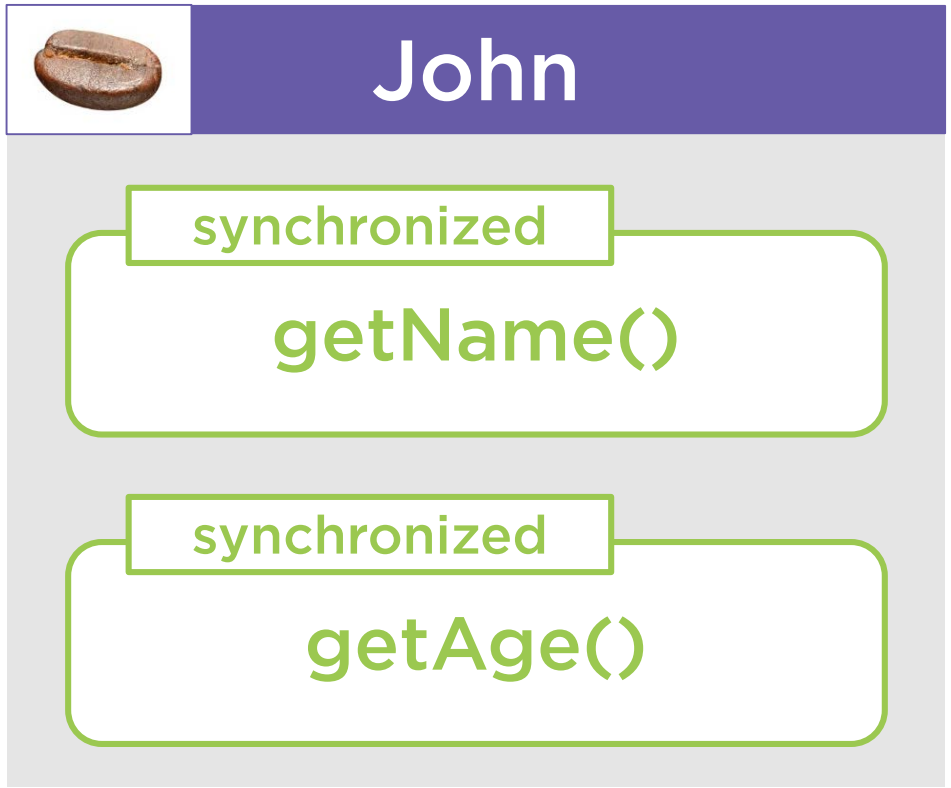
Synchronizing More Than One Method



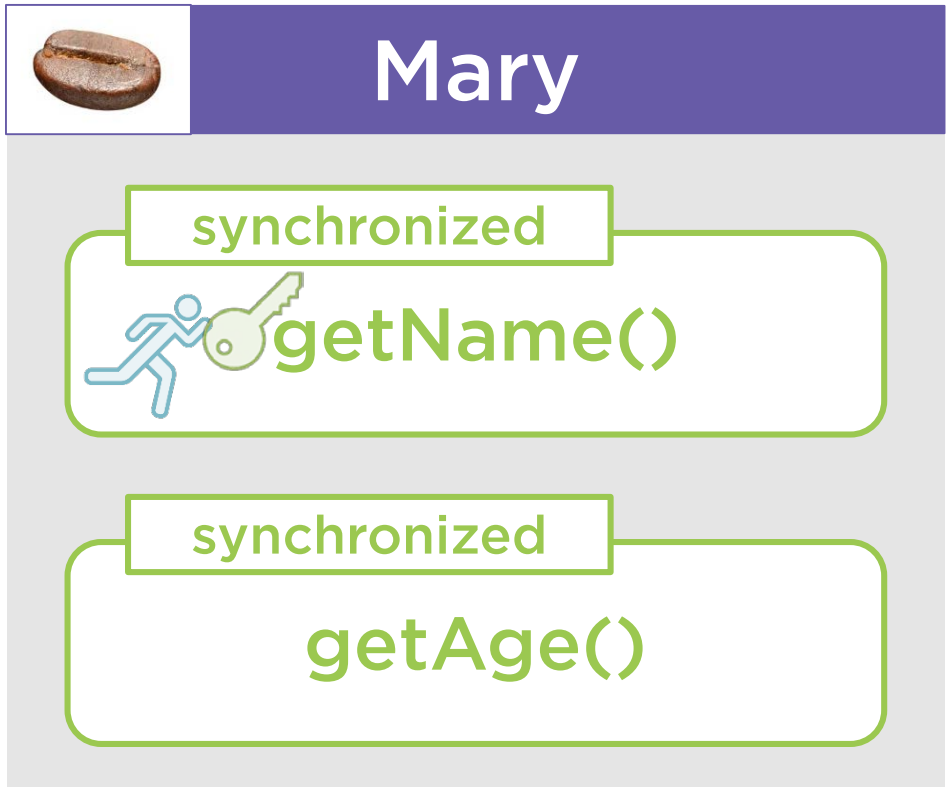


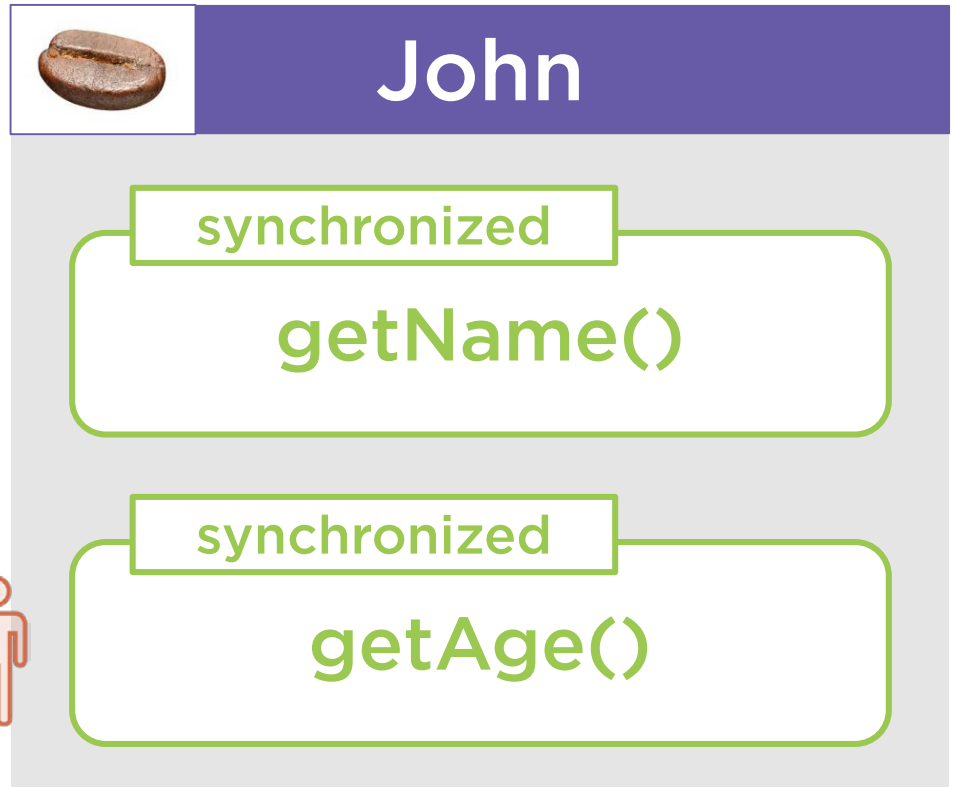
Synchronizing More Than One Method



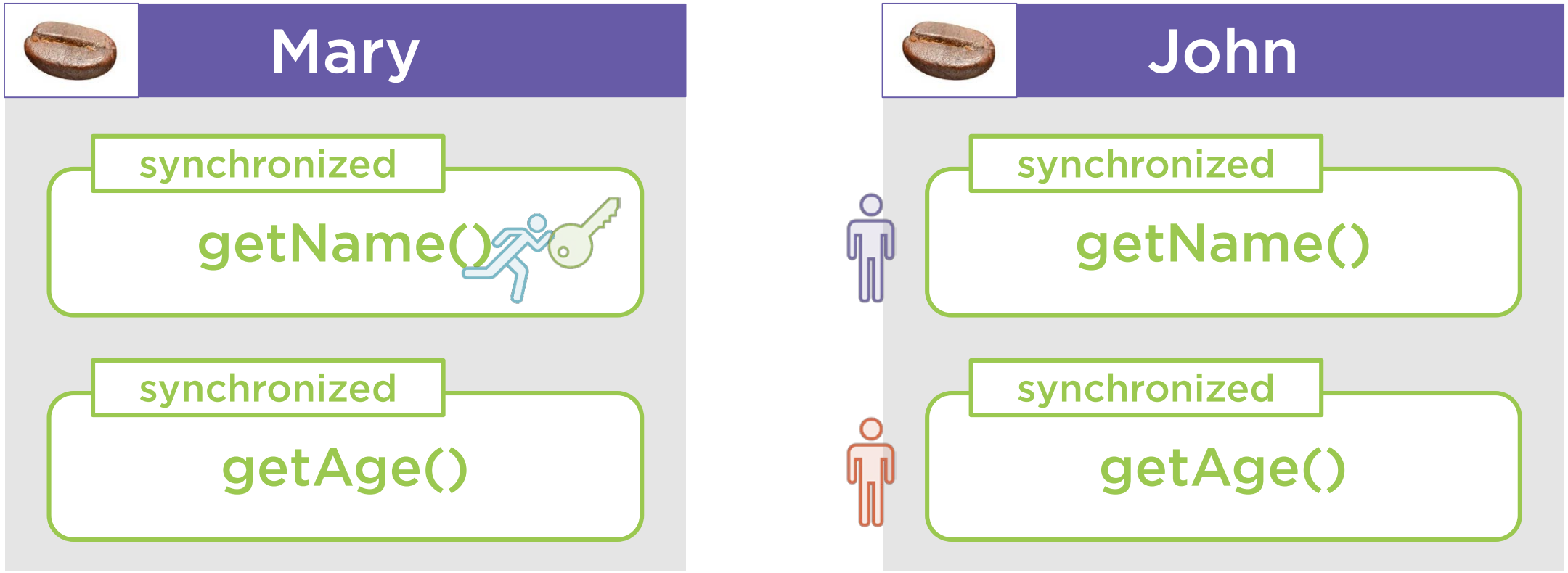


Synchronizing More Than One Method



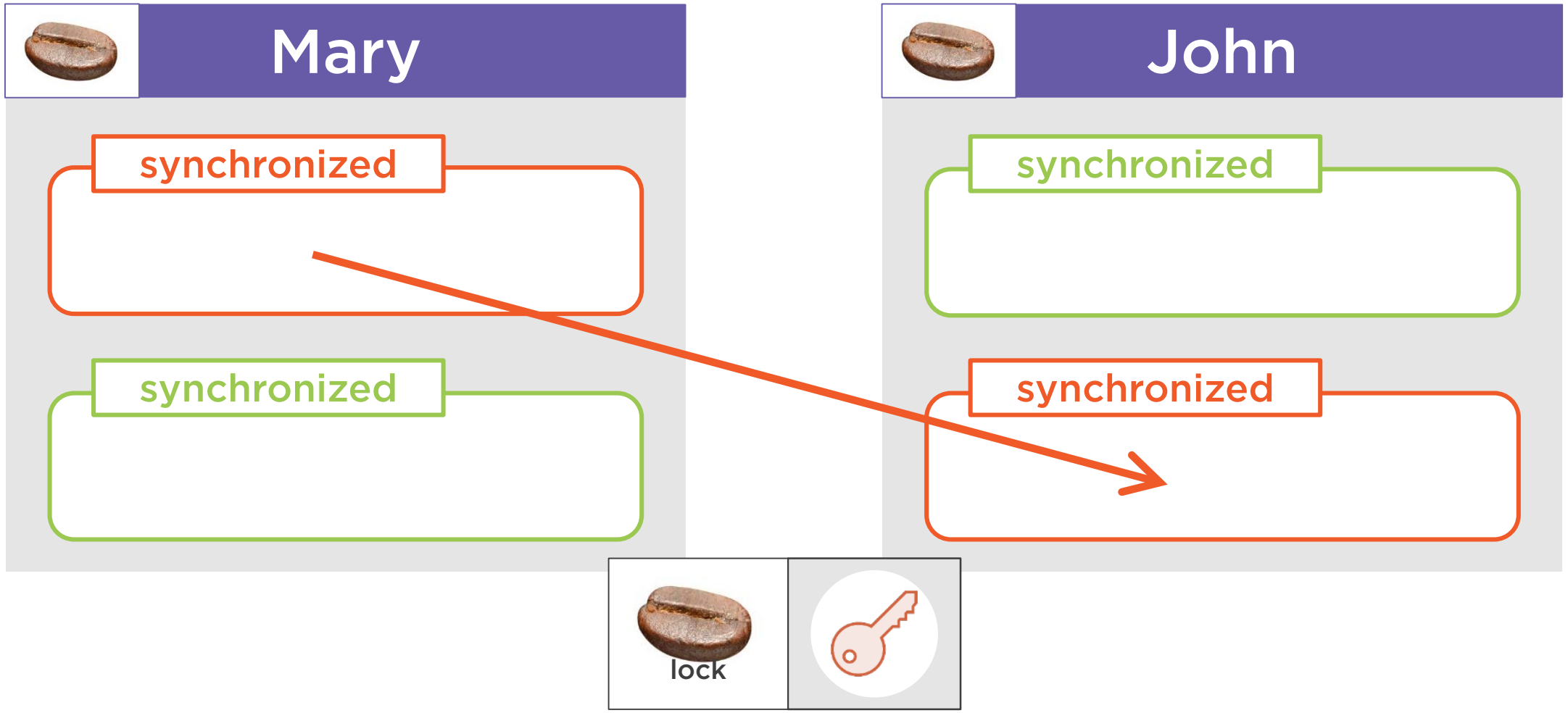


Synchronizing More Than One Method

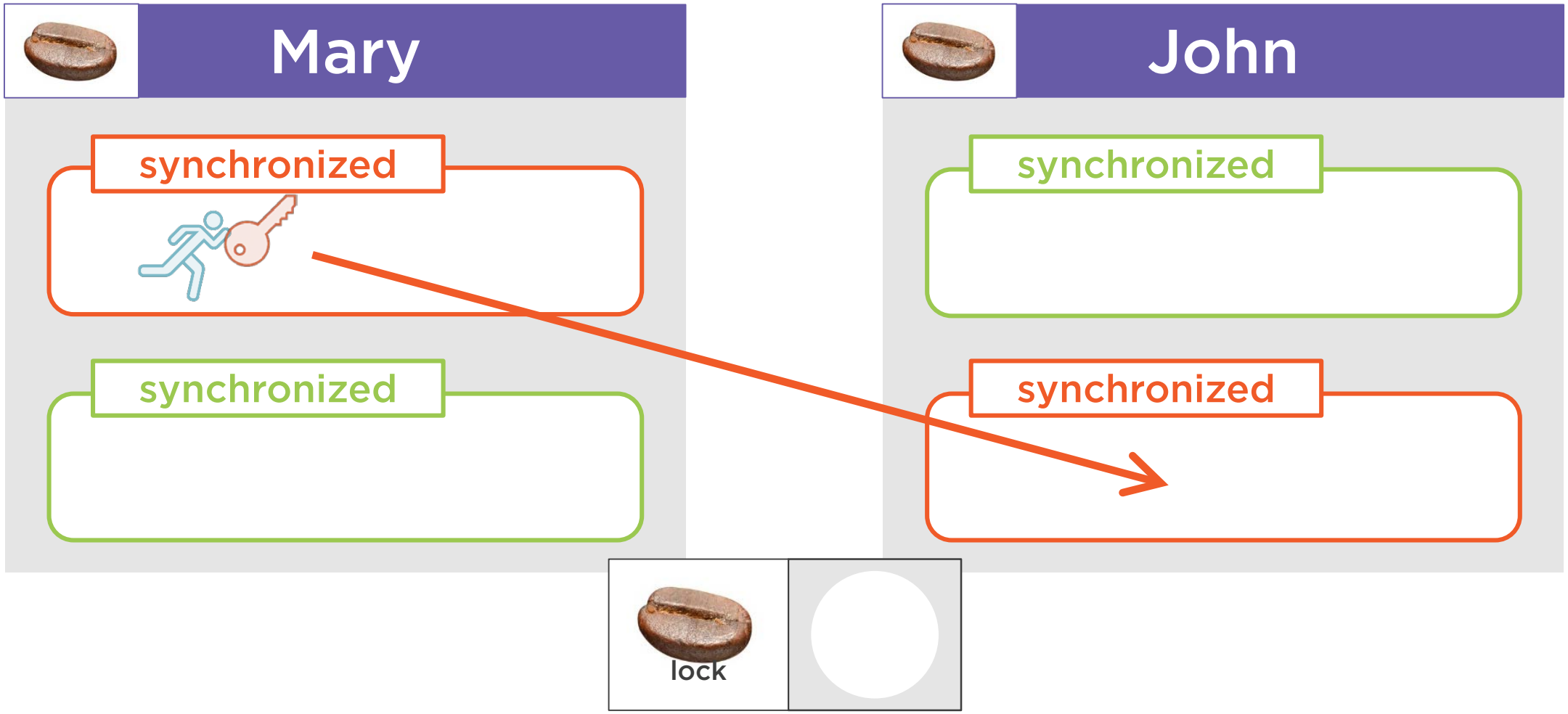


Reentrant Locks and Deadlocks

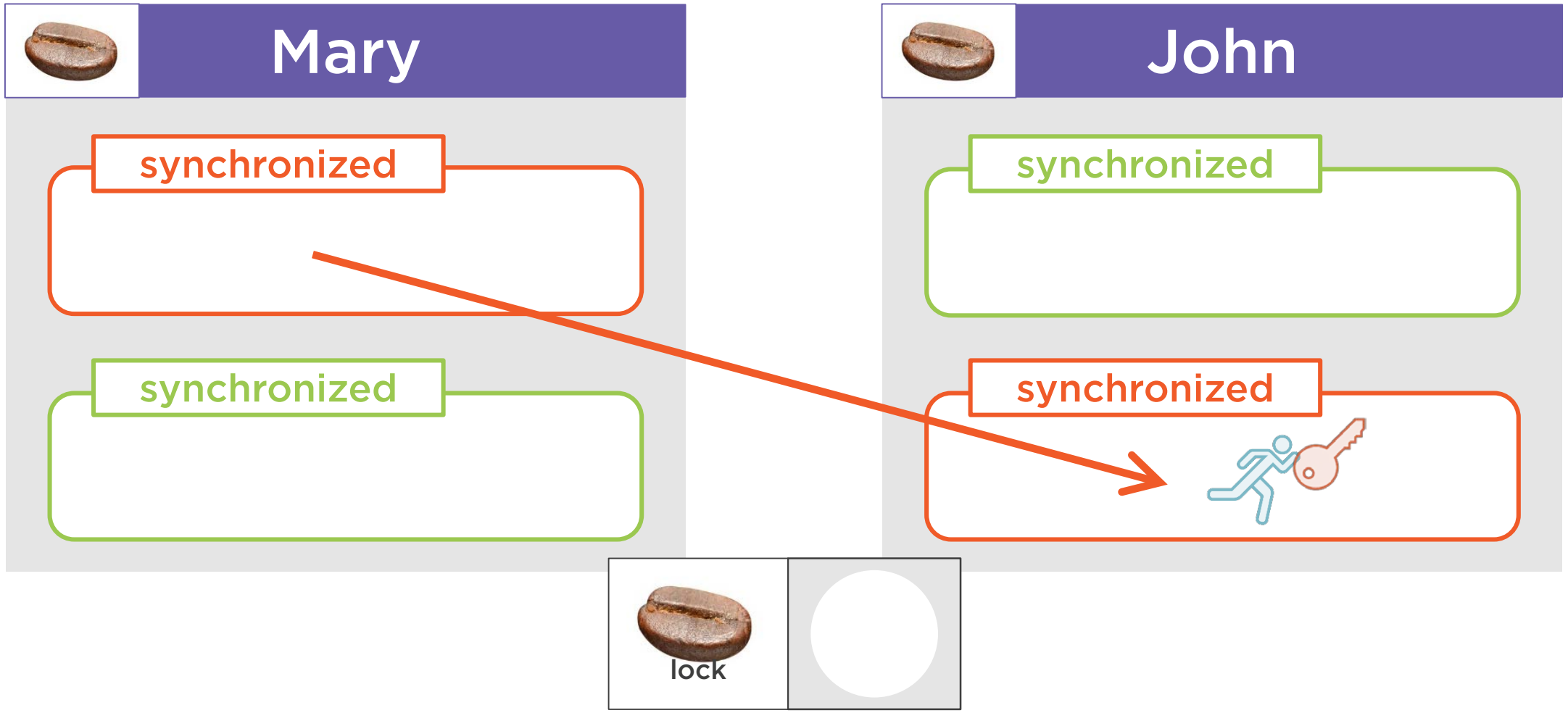
Locks Are Reentrant



Locks Are Reentrant



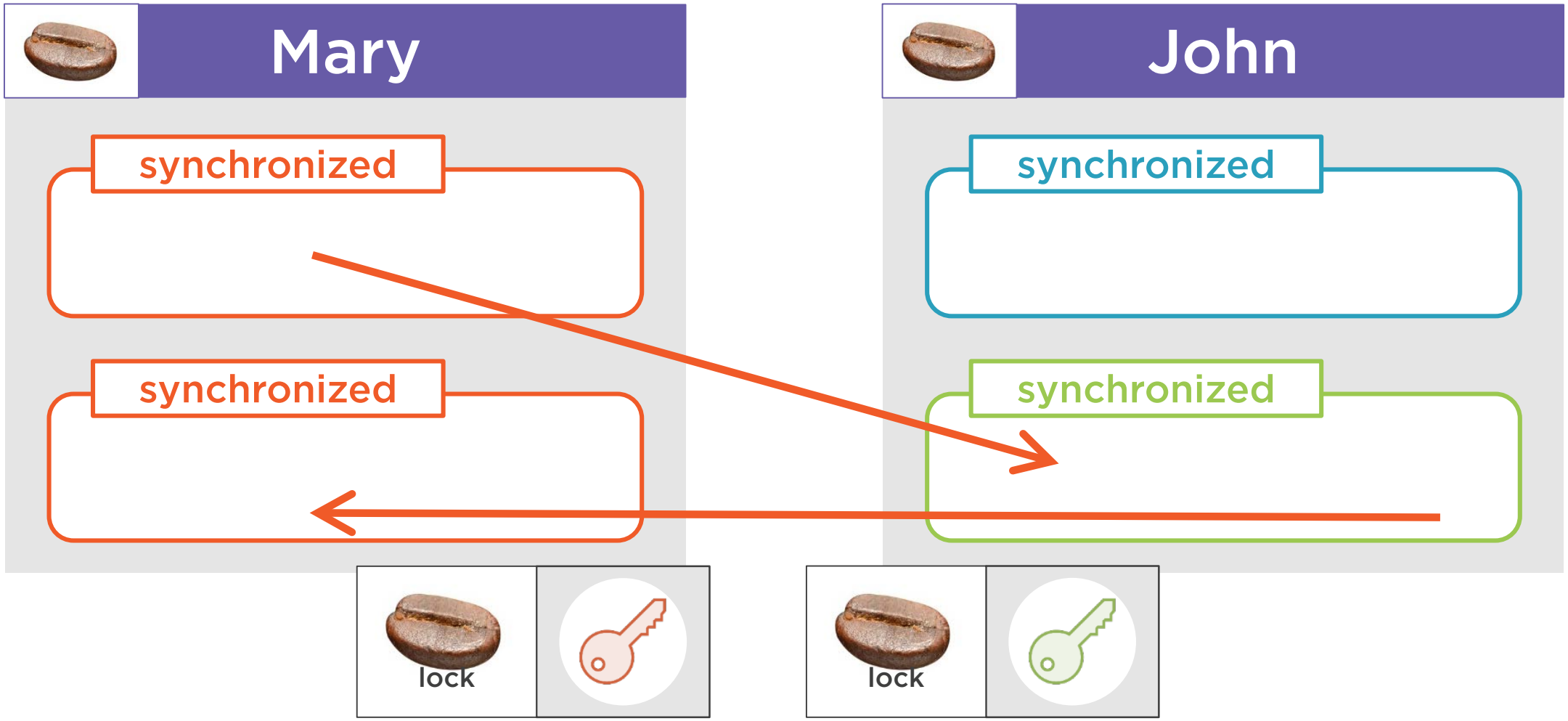
Locks Are Reentrant



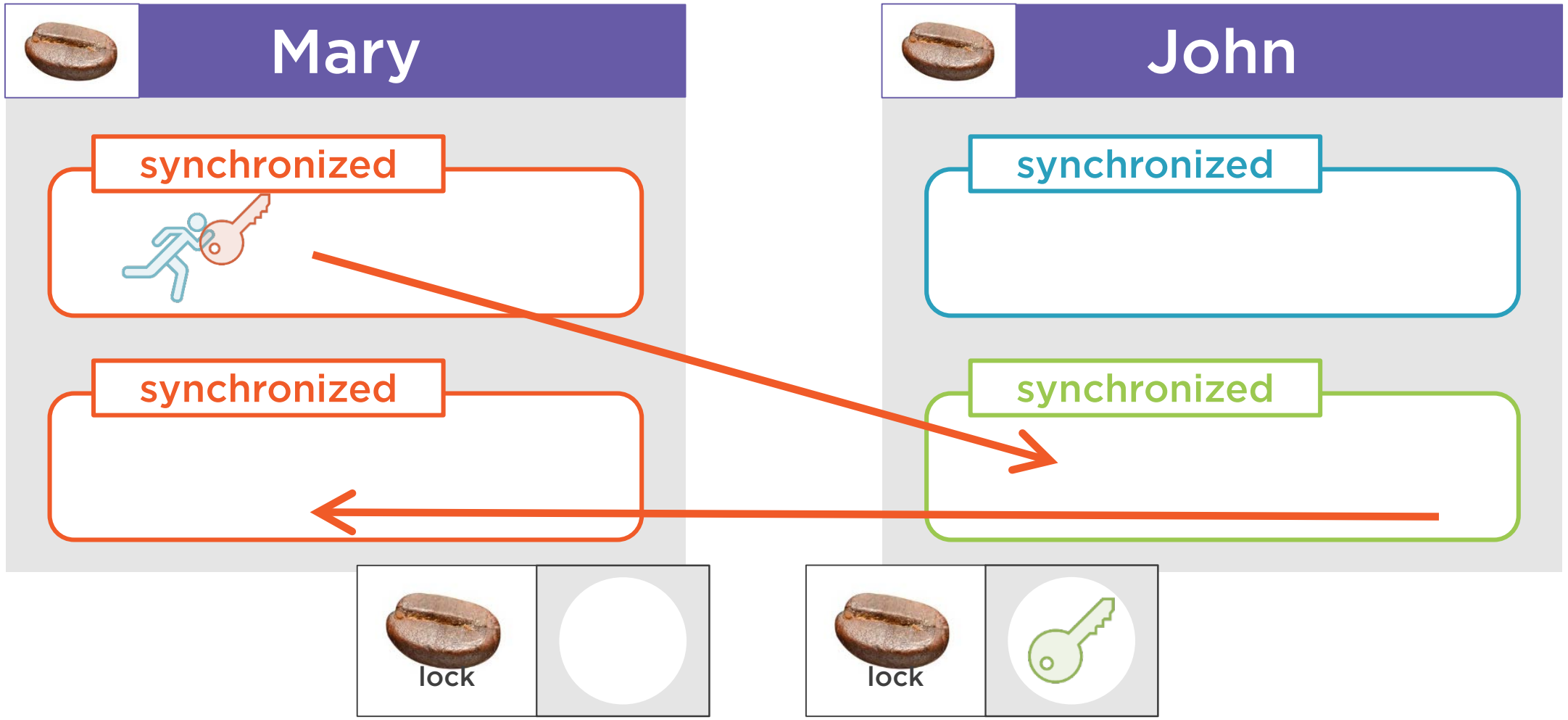
Locks are reentrant

When a thread holds a lock, it can enter a block synchronized on the lock it is holding

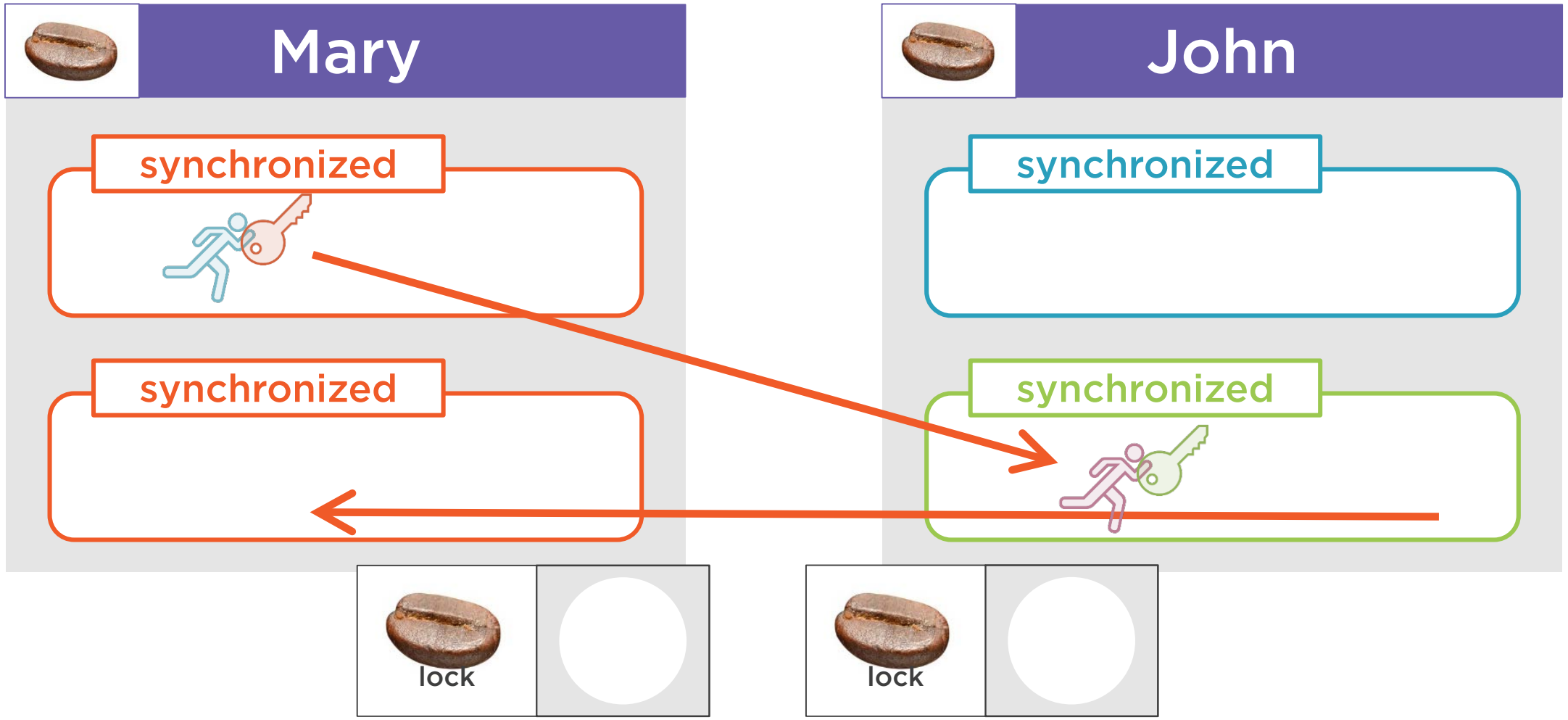
Deadlocks


Mary
John
synchronized
synchronized
synchronized
synchronized
lock
lock

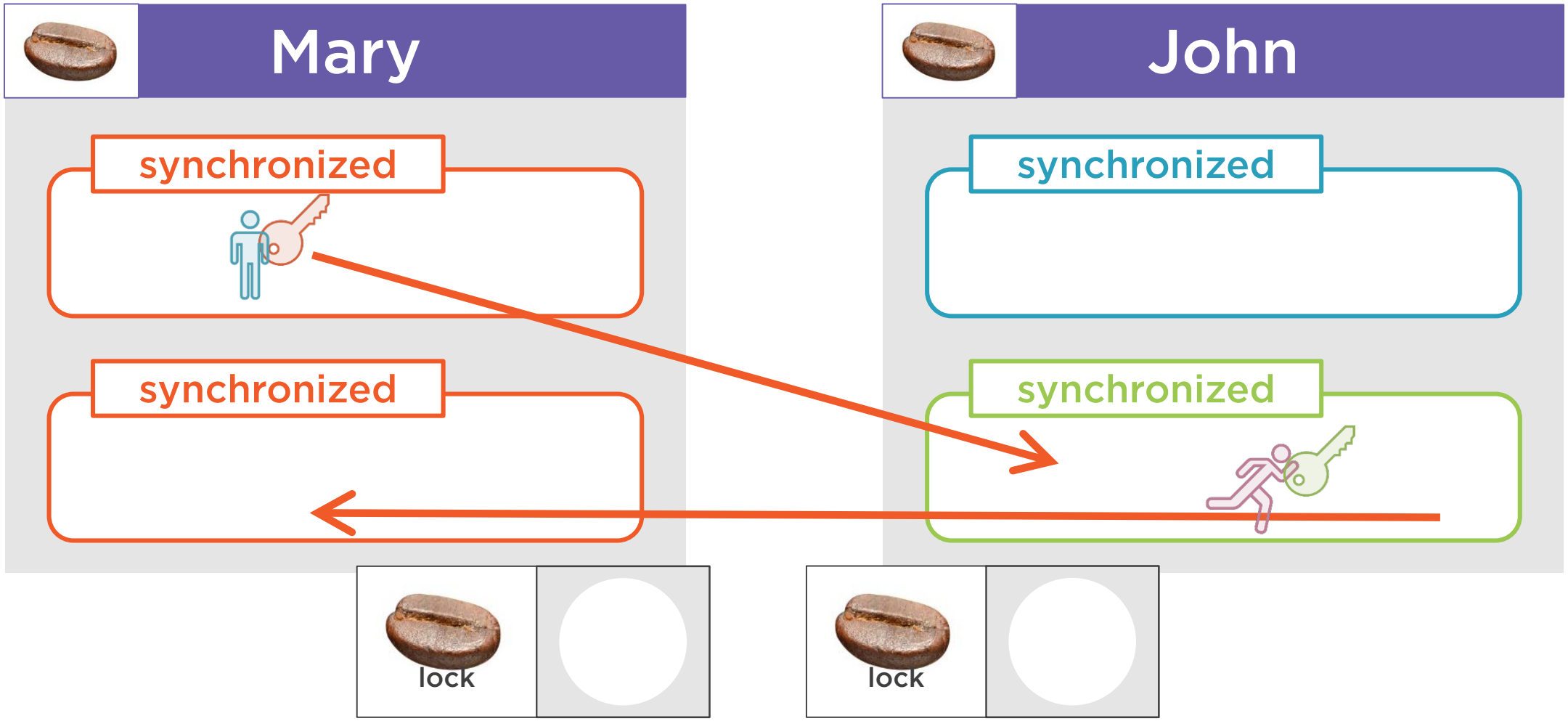
Deadlocks


Mary
John
synchronized
synchronized
synchronized
synchronized
lock
lock

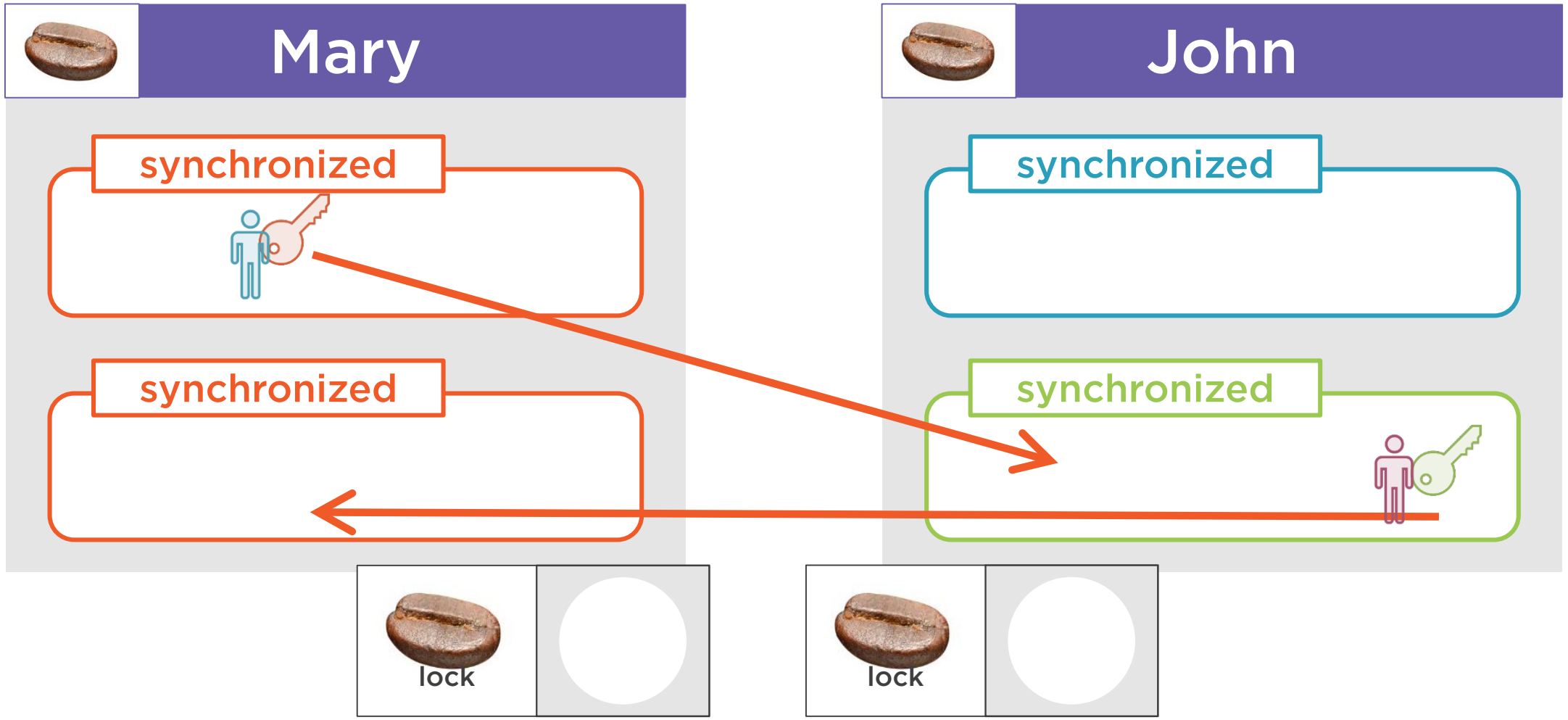
Deadlocks


Mary
John
synchronized
synchronized
synchronized
synchronized
lock
lock

Deadlocks

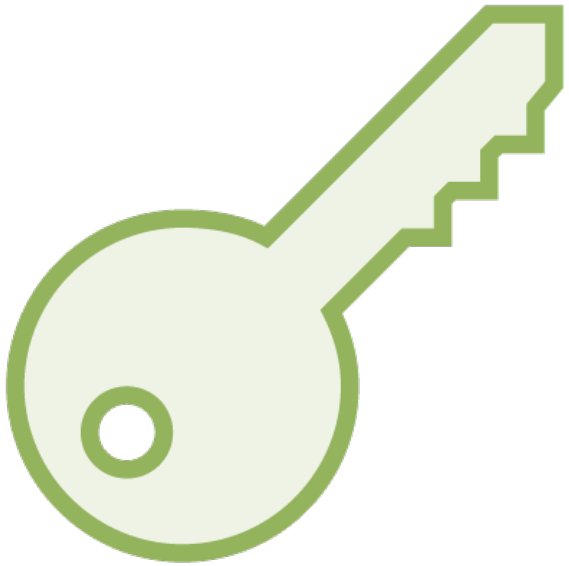

Mary
John
synchronized
synchronized
synchronized
synchronized
lock
lock

Deadlocks


Mary
John
synchronized
synchronized
synchronized
synchronized
lock
lock

Deadlock

A deadlock is a situation where a thread T_1 holds a key needed by a thread T_2,
and T_2 holds the key needed by T_1



The JVM is able to detect deadlock situations, and can log information to help debug the application

But there is not much we can do if a deadlock situation occurs, beside rebooting the JVM

A First Glimpse at the Runnable Pattern

The most basic way to create threads in Java is to use the Runnable Pattern

First create an instance of Runnable

Then pass it to the constructor of the Thread class

Then call the start() method of this thread object

```
Runnable runnable = new Runnable() {
    public void run() {
        String name = Thread.currentThread().getName();
        System.out.println("I am running in thread " + name);
    }
}
```

First create an instance of a Runnable

This is the Java 7 way, with an instance of an anonymous class

```
Runnable runnable = () -> {

        String name = Thread.currentThread().getName();
        System.out.println("I am running in thread " + name);

}
```

First create an instance of a Runnable

This is the Java 8 way, with a lambda expression

```
Thread thread = new Thread(runnable);
thread.start();
```

Second, pass it to the constructor of the Thread class

Third start the thread!

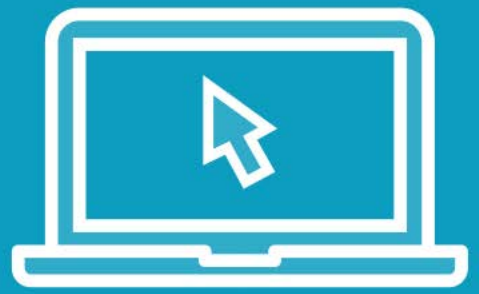
```
Thread thread = new Thread(runnable);
thread.start();
```

Second, pass it to the constructor of the Thread class

Third start the thread!

This is all we need to know for now...

Demo



Let us see some code!

Let us create threads on simple examples

See what can go wrong with a race condition

Fix our code with synchronization

Wrapup

What did we learn?

A thread executes a task in a special context

The fundamental notion of race condition

How to synchronize code to avoid race conditions

Reentrance and deadlocks

How to use a debugger to analyze threads and sort out a deadlock situation