Multithreading 1: Atomic/ Volatile 2. ThreadLocal

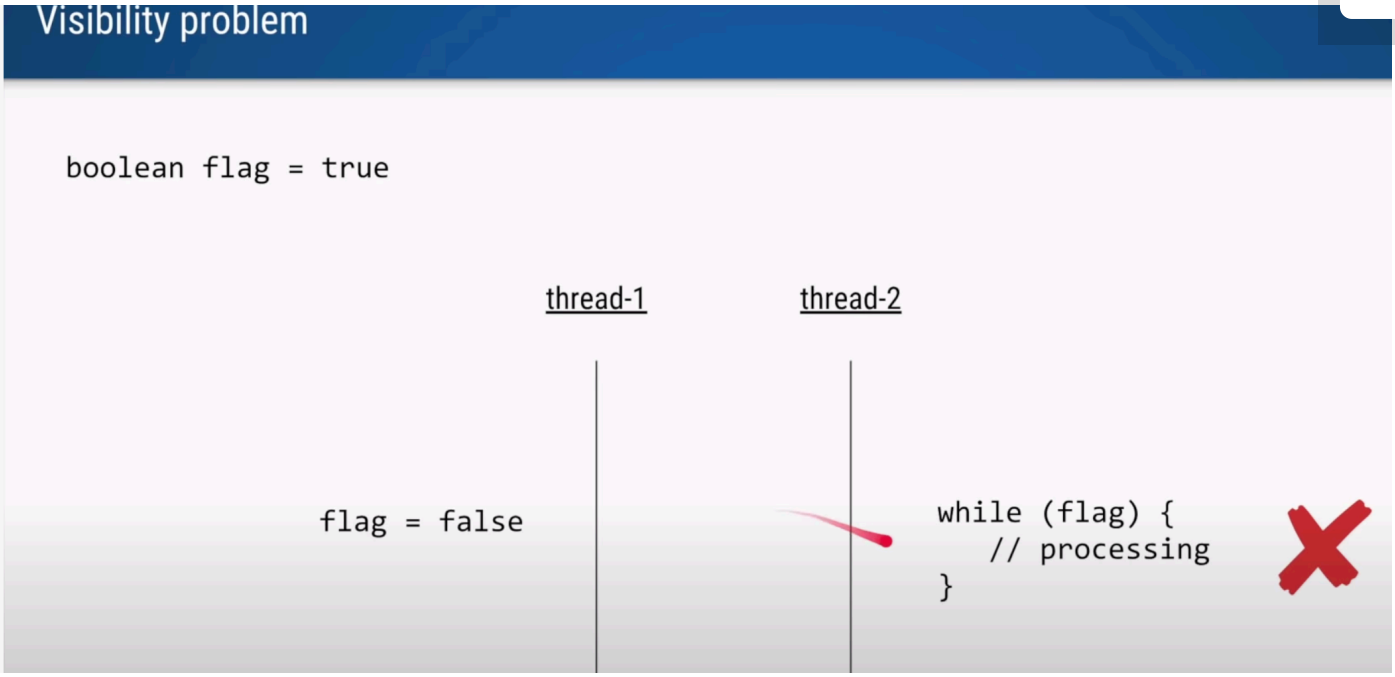
Not complete

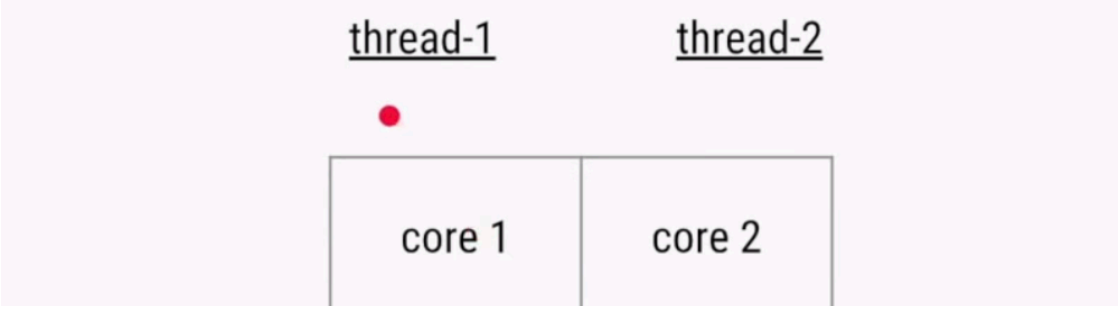
Topic 1:

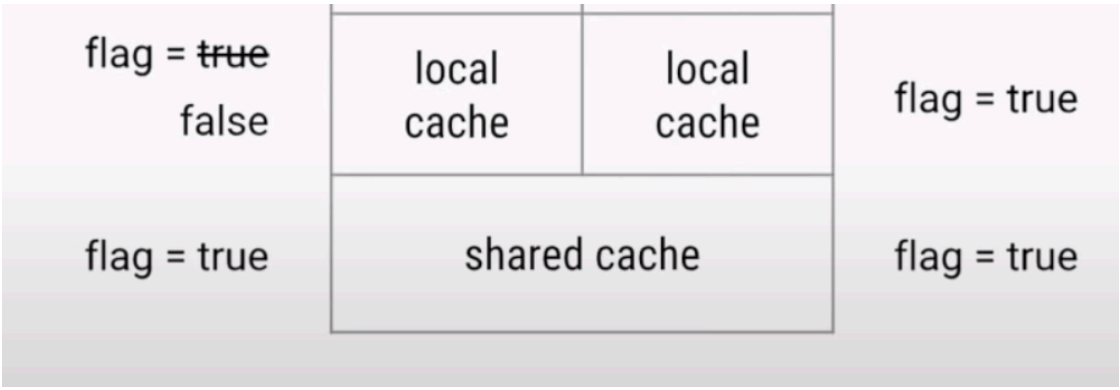
Atomic Vs Volatile:

Volatile: It is used to solve Visibility problem across multipal thread.

The Java volatile keyword is used to mark a Java variable as "being stored in main memory". More precisely that means, that every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache, and that every write to a volatile variable will be written to main memory, and not just to the CPU cache.

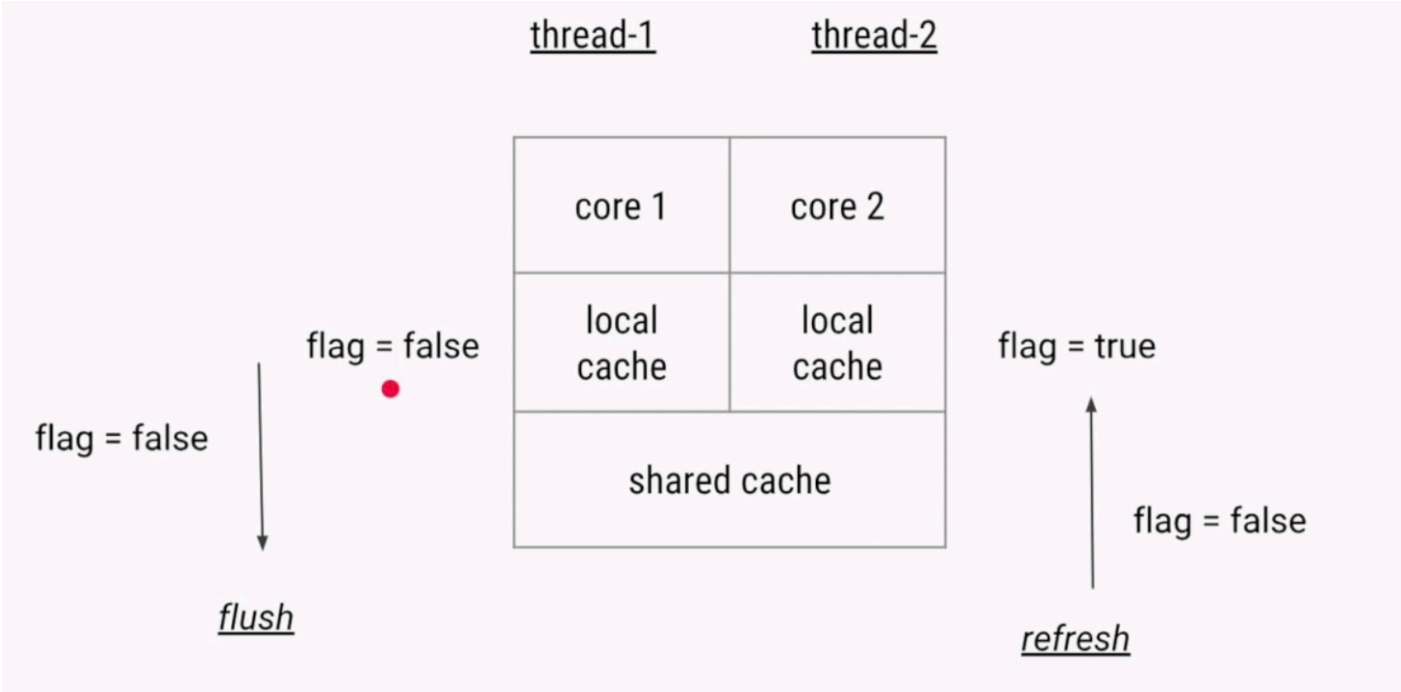




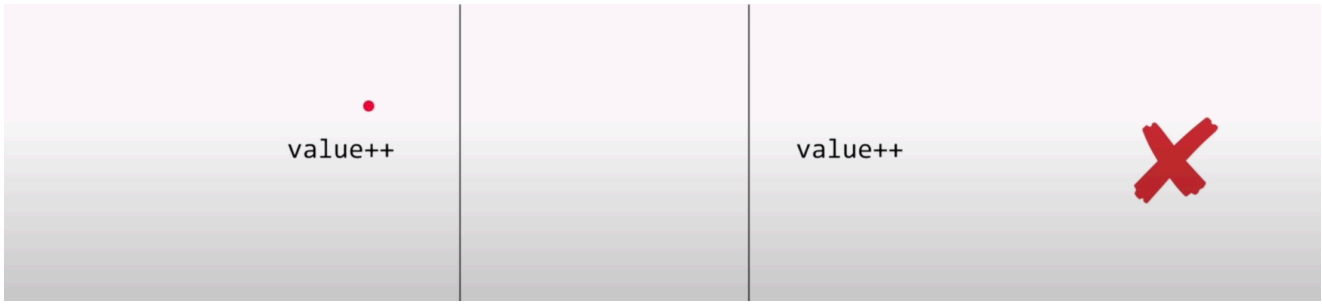


After using volatile keyword:

Any update will reflect will other thread as it flush the changes in shared cache pool.



SYNCHRONIZATION PROBLEM:



There are 2 op : read and write

Synchronization problem

```
volatile int value = 1;
```

Even with volatile

#	Thread-1	Thread-2
1	Read value (=1)	
2		Read value (=1)
3	Add 1 and write (=2)	
4		Add 1 and write (=2)

volatile won't work here bcoz even after step 2 value is 1 in both cases.

To solve this we need to have **atomic operation** or **synchronized block.**

Sol 1:

Synchronization solutions - #1

```
volatile int value = 1;
```

thread-1 thread-2

```
synchronized (obj) {
```

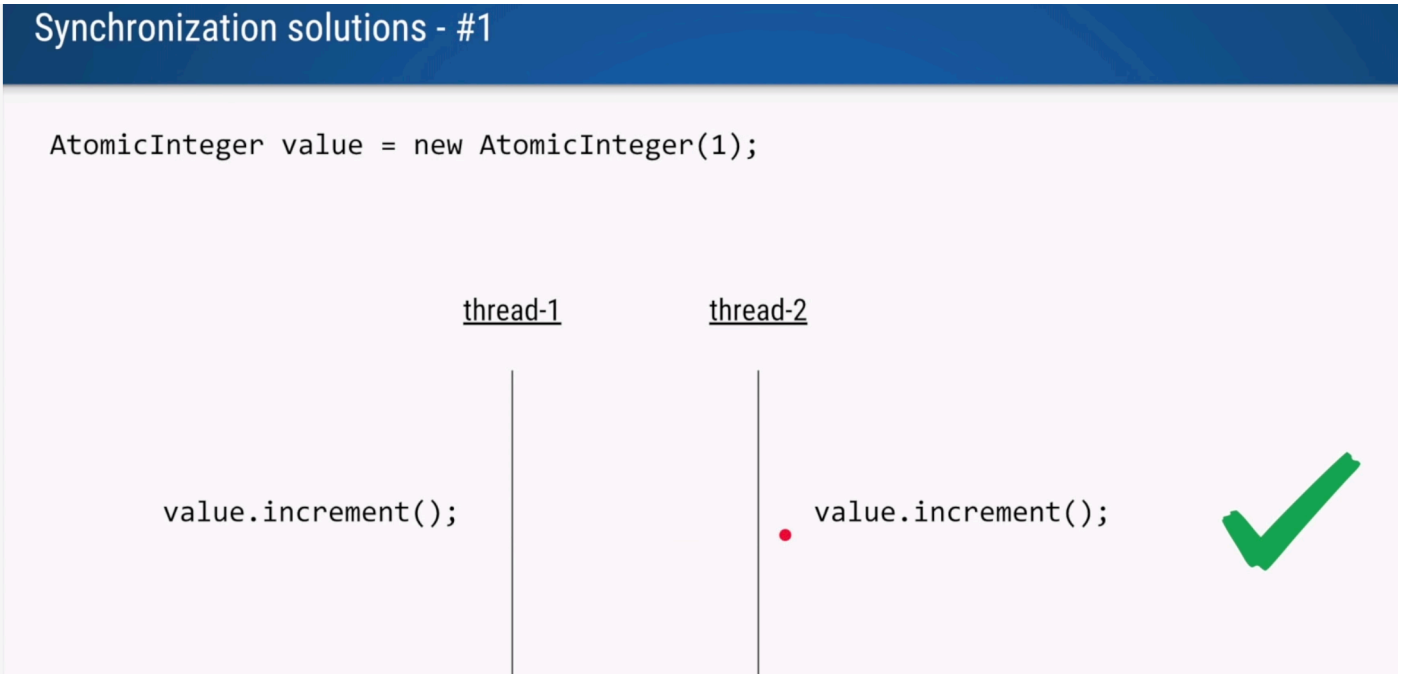
```
synchronized (obj) {
```

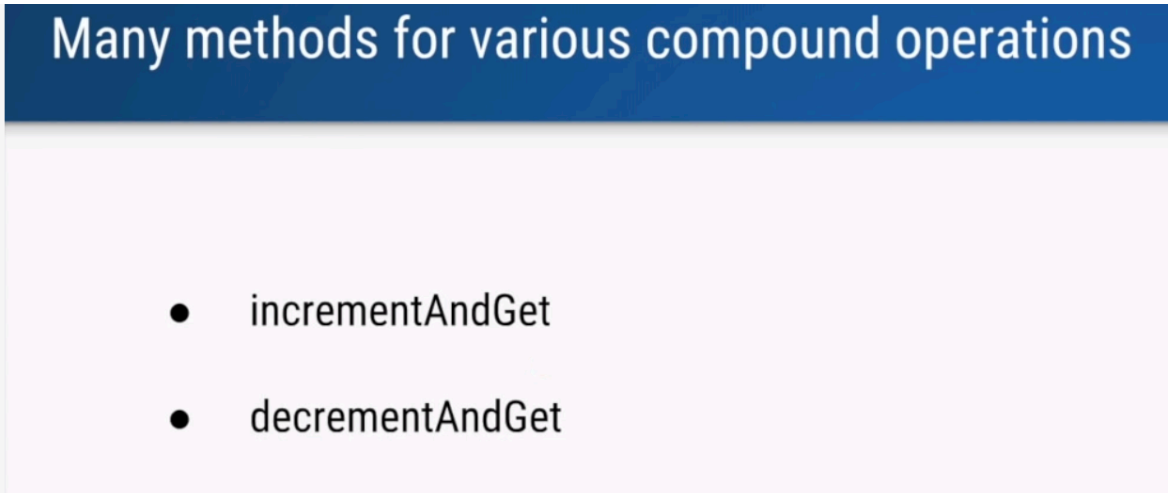
only one thread can update synchronized block at a time.

Sol2:

Using Atomic variable:

The primary use of AtomicInteger is when we are in multi-threaded context and we need to perform atomic operations on an int value without using synchronized keyword.





- addAndGet (int delta)
- compareAndSet (int expectedValue, int newValue)

So If we have a visibility problem then need to go with Volatile and If we have a compound operation and we want to do that atomically then we should go with AtomicInteger.

Typical Use Cases

Type	Use Case
volatile	Flags
AtomicInteger AtomicLong	Counters
AtomicReference	Caches (building new cache in background and replacing atomically) Used by some internal classes Non-blocking algorithms

Reference : http://tutorials.jenkov.com/java-concurrency/volatile.html. BEST

https://howtodoinjava.com/java/multi-threading/atomicinteger-example/ howtodoinjava

Topic 2:

ThreadLocal:

It provides per thread object

Most common use of thread local is when you have some object that is not thread-safe, but you want to avoid synchronizing access to that object using synchronized keyword/block. Instead, give each thread its own instance of the object to work with.

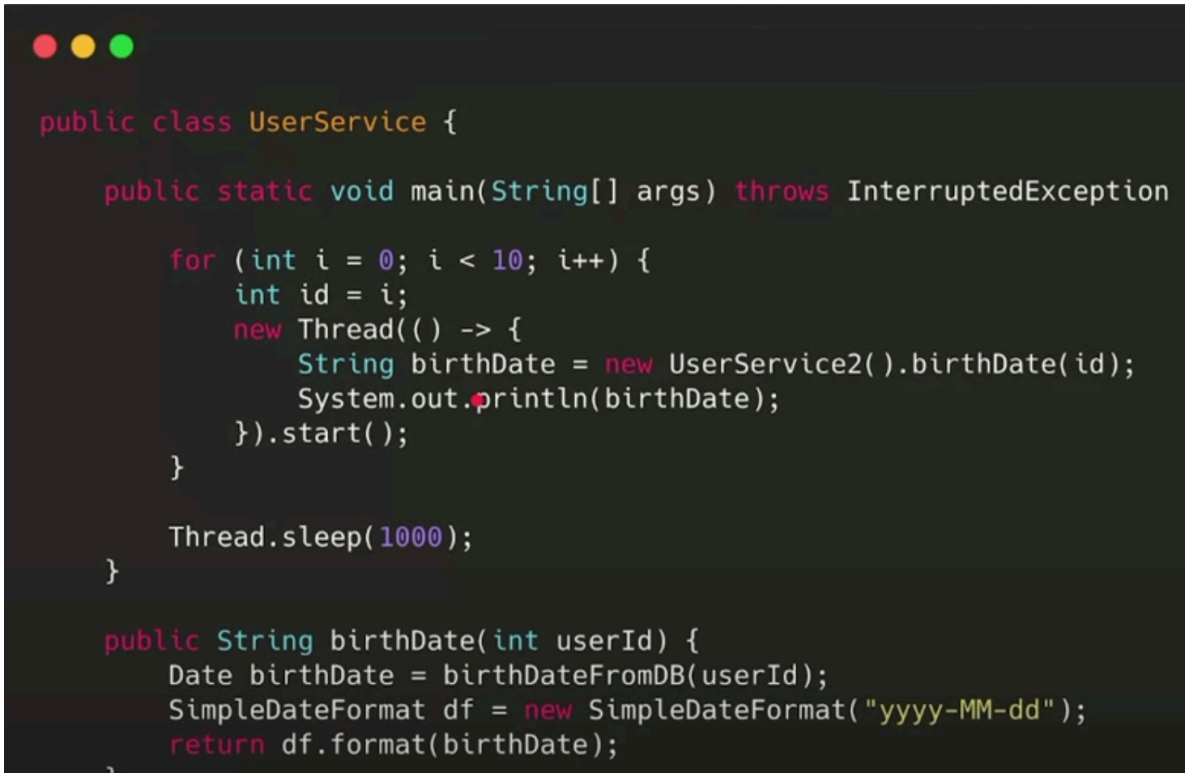
A good alternative to synchronization or threadlocal is to make the variable a local variable. Local variables are always thread safe. The only thing which may prevent you to do this is your application design constraints.

Note : In wabapp server, it may be keep a thread pool, so a ThreadLocal var should be removed before response to the client, since current thread may be reused by next request. Also, if you do not clean up when you're done, any references it holds to classes loaded as part of a deployed webapp will remain in the permanent heap and will never get garbage collected.

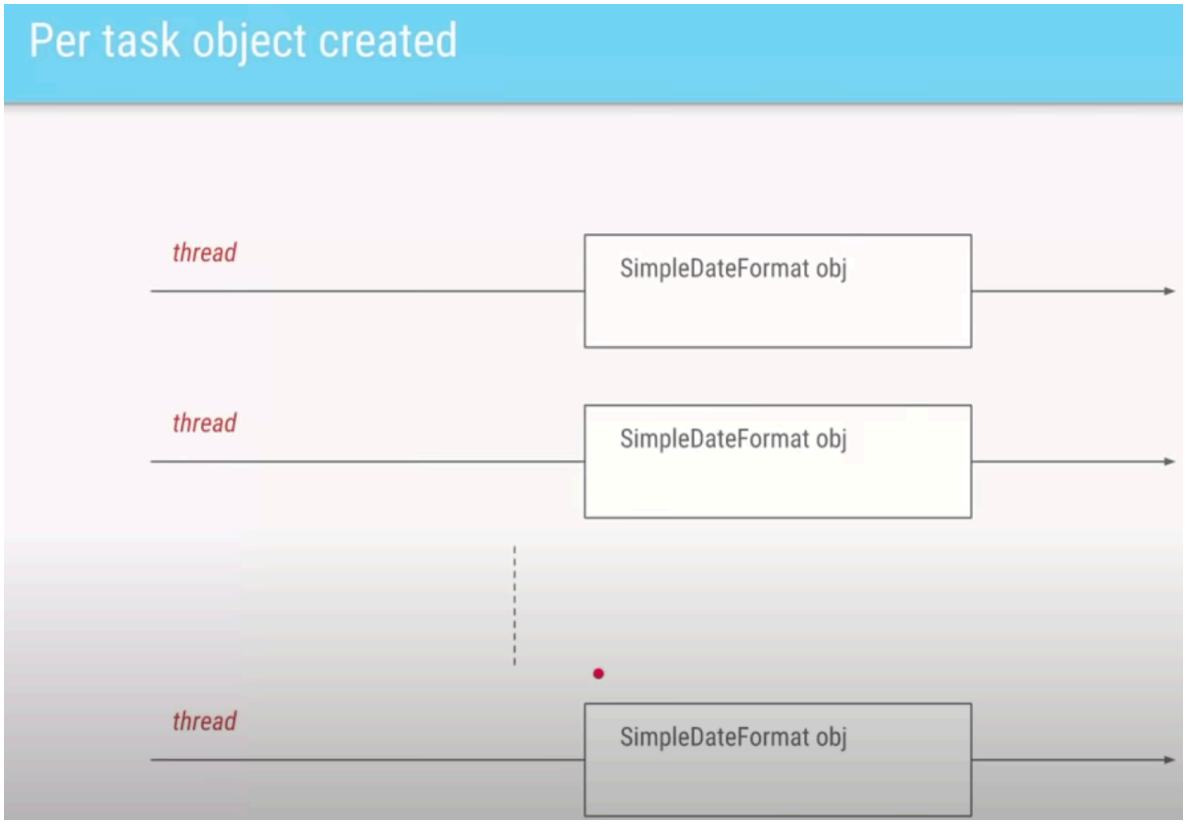
```
private ThreadLocal threadLocal = new ThreadLocal();
```

This only needs to be done once per thread. Multiple threads can now get and set values inside this ThreadLocal, and each thread will only see the value it set itself.

USeCase1:



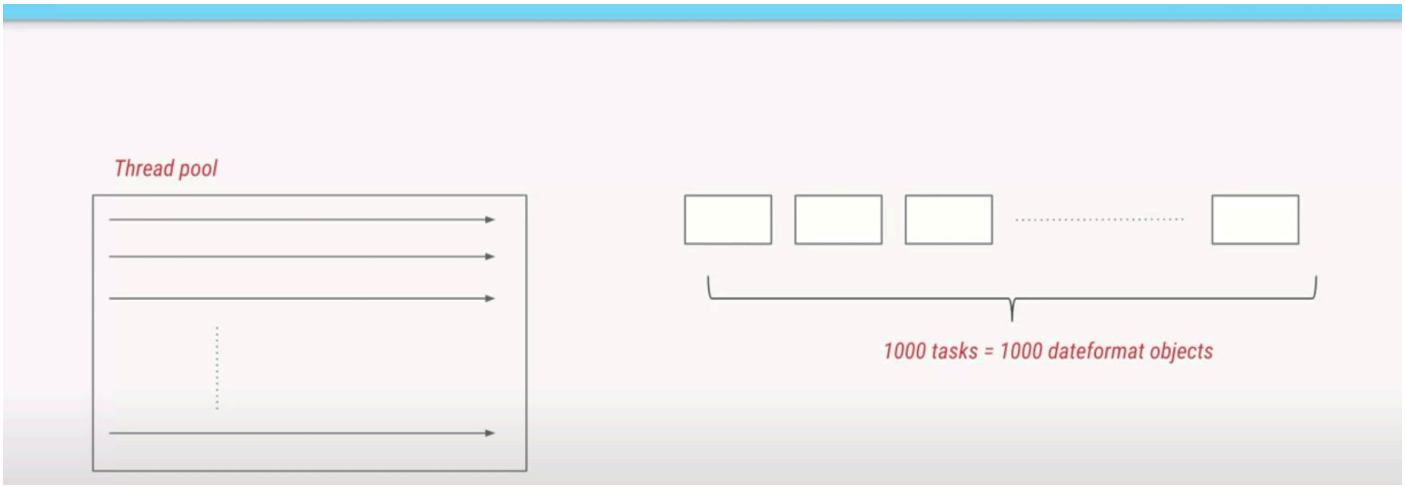
```
    }

    // ......
}
```



To solve this and prevent creating so many thread we use executor service:

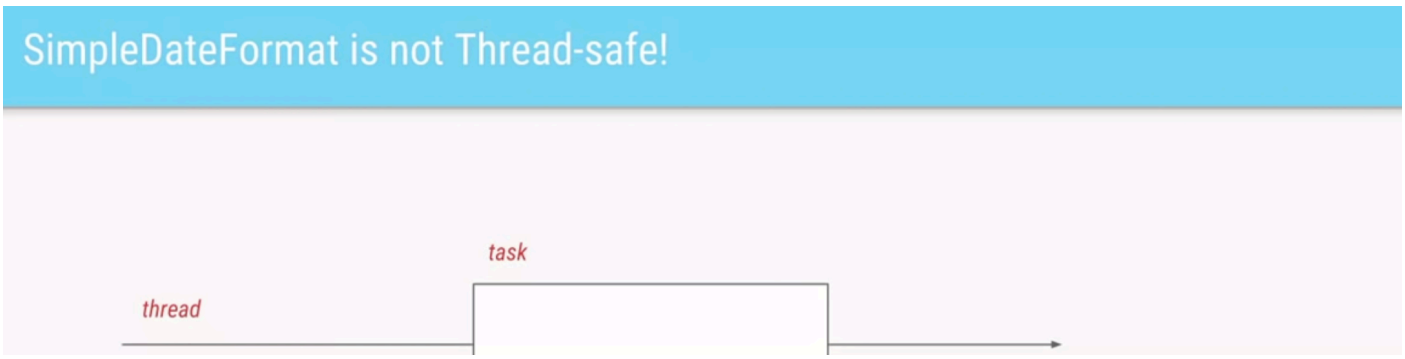
```
private static ExecutorService threadPool = Executors.newFixedThreadPool(10);

public static void main(String[] args) throws InterruptedException {

    for (int i = 0; i < 1000; i++) {
        int id = i;
        threadPool.submit(() -> {
            String birthDate = new UserService2().birthDate(id);
            System.out.println(birthDate);
        });
    }

    Thread.sleep(1000);
}
```

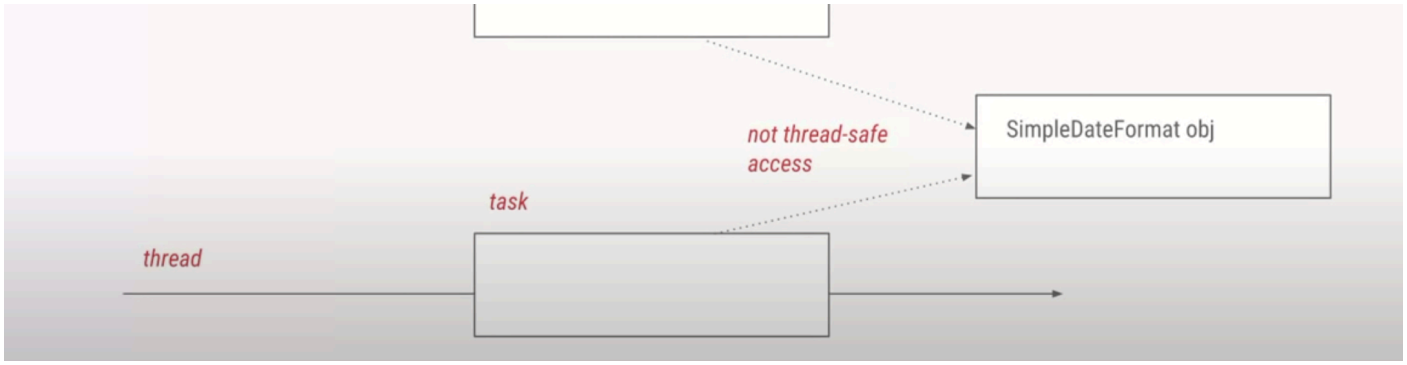
1000 objects created



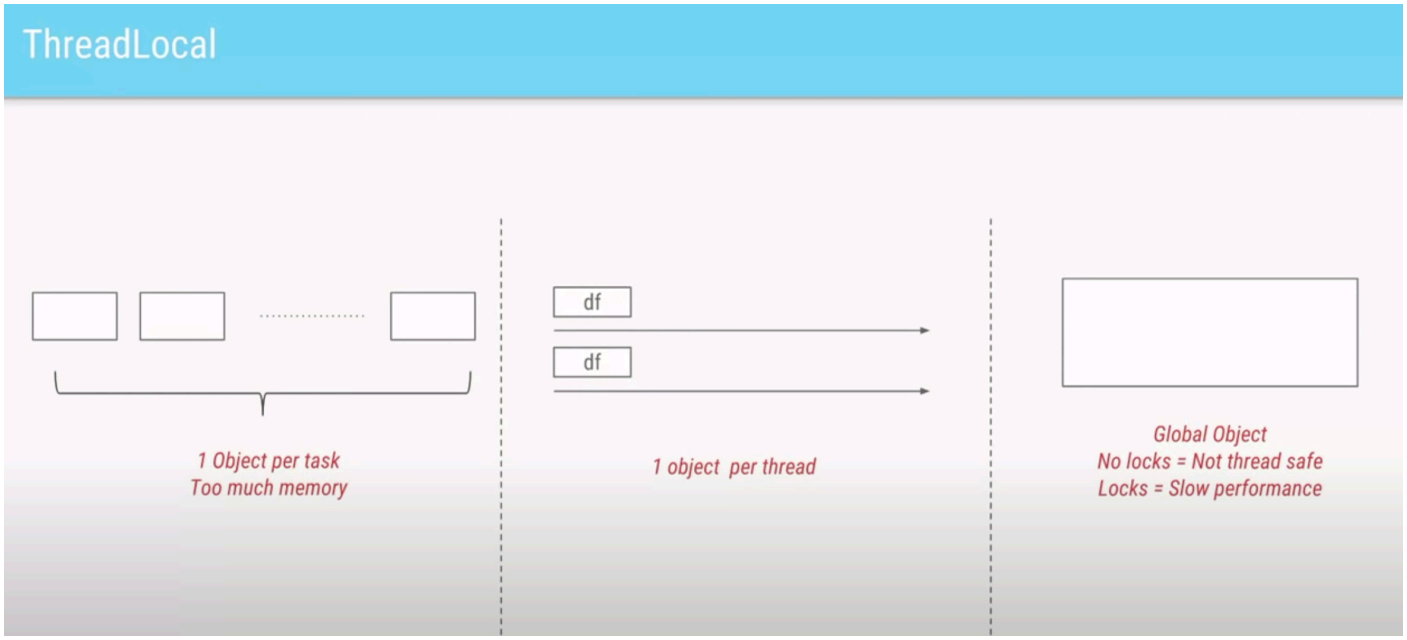
To prevent for creating too many dataformat objects , we could cry eate a global variable like:

```
private static ExecutorService threadPool = Executors.newFixedThreadPool(10);
private static SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd");

public static void main(String[] args) throws InterruptedException {

    for (int i = 0; i < 1000; i++) {
        int id = i;
        threadPool.submit(() -> {
            String birthDate = new UserService2().birthDate(id);
            System.out.println(birthDate);
        });
    }

    Thread.sleep(1000);
}

public String birthDate(int userId) {
    Date birthDate = birthDateFromDB(userId);
    return df.format(birthDate);
}
```

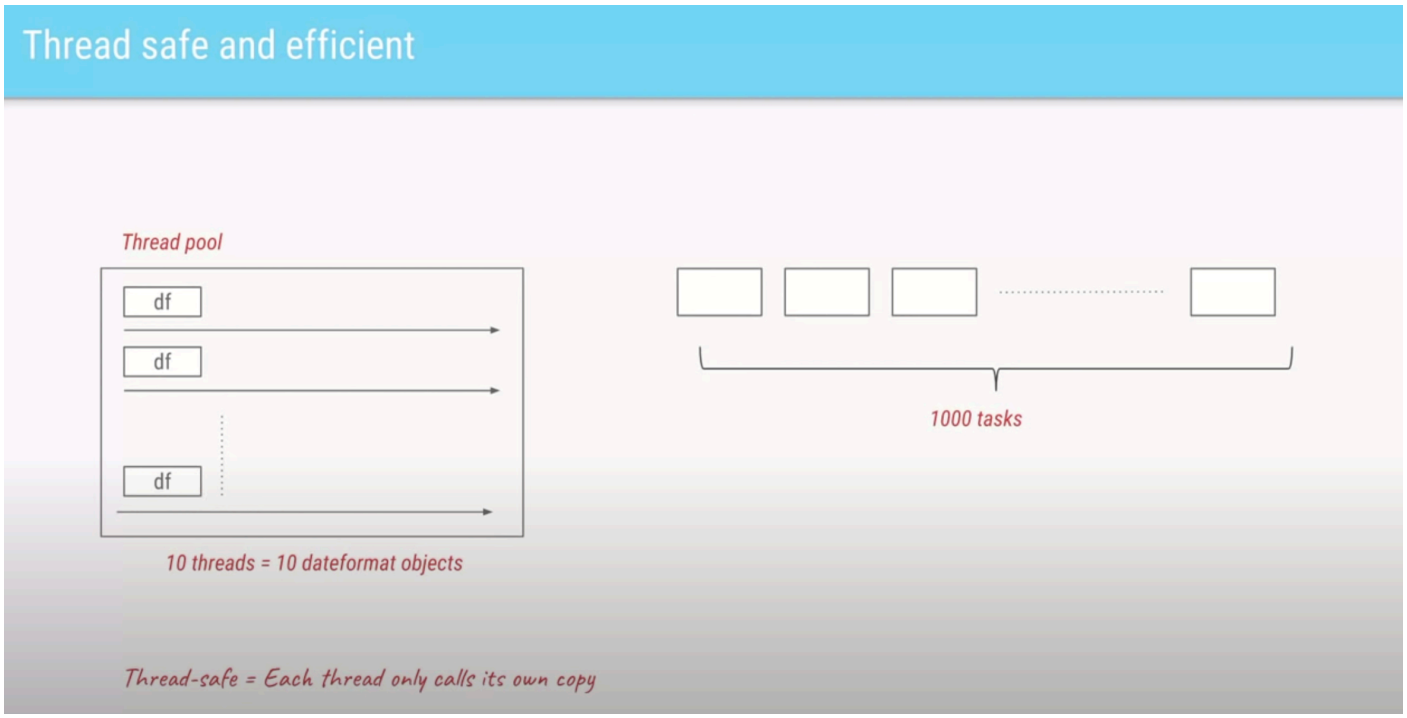




to solve this we could have object for a thread only.



So it will have per thread instance and also it will provide thread safety



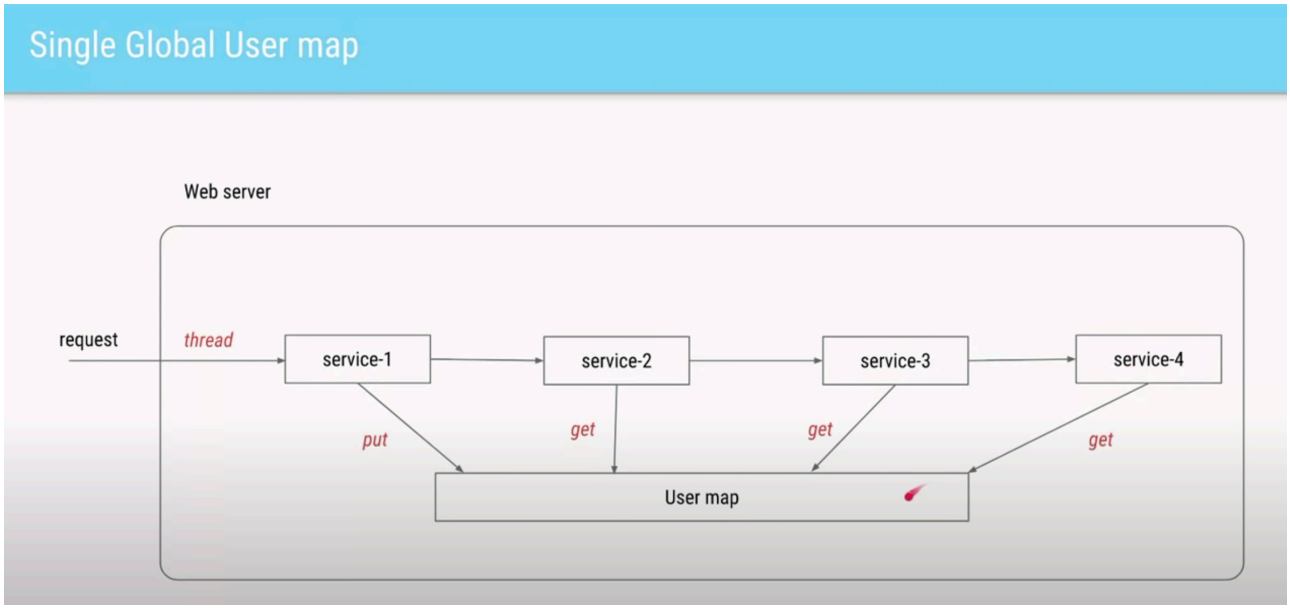
```
class ThreadSafeFormatter {

    public static ThreadLocal<SimpleDateFormat> dateFormatter = new ThreadLocal<SimpleDateFormat>(){
        @Override
        protected SimpleDateFormat initialValue( ) {
            return new SimpleDateFormat("yyyy-MM-dd");
        }
        @Override
        public SimpleDateFormat get( ) {
            return super.get( );
        }
    };
}

public class UserService {

    public static void main(String[] args) {
        // ....
    }

    public String birthDate(int userId) {
        Date birthDate = birthDateFromDB(userId);
        final SimpleDateFormat df = ThreadSafeFormatter.dateFormatter.get( );
        return df.format(birthDate);
    }
}
```

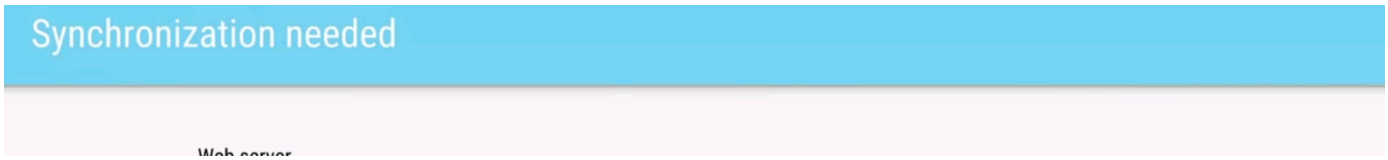
Called once for each thread

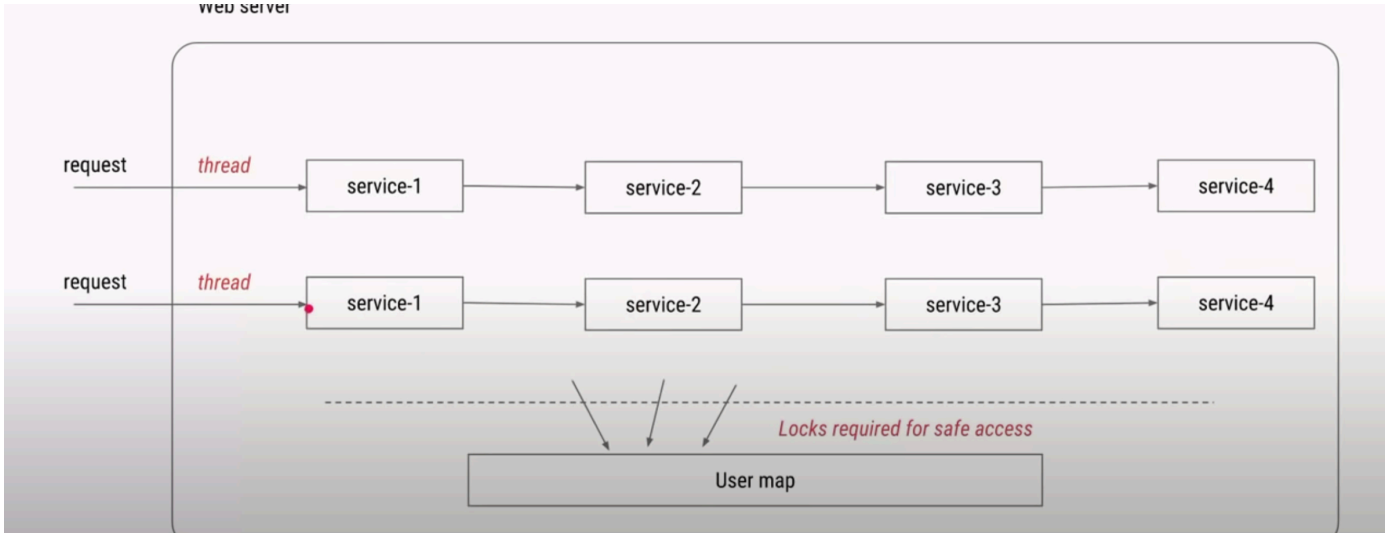
1st call = initialValue()
Subsequent calls will return same initialized value

Each thread will get its own copy

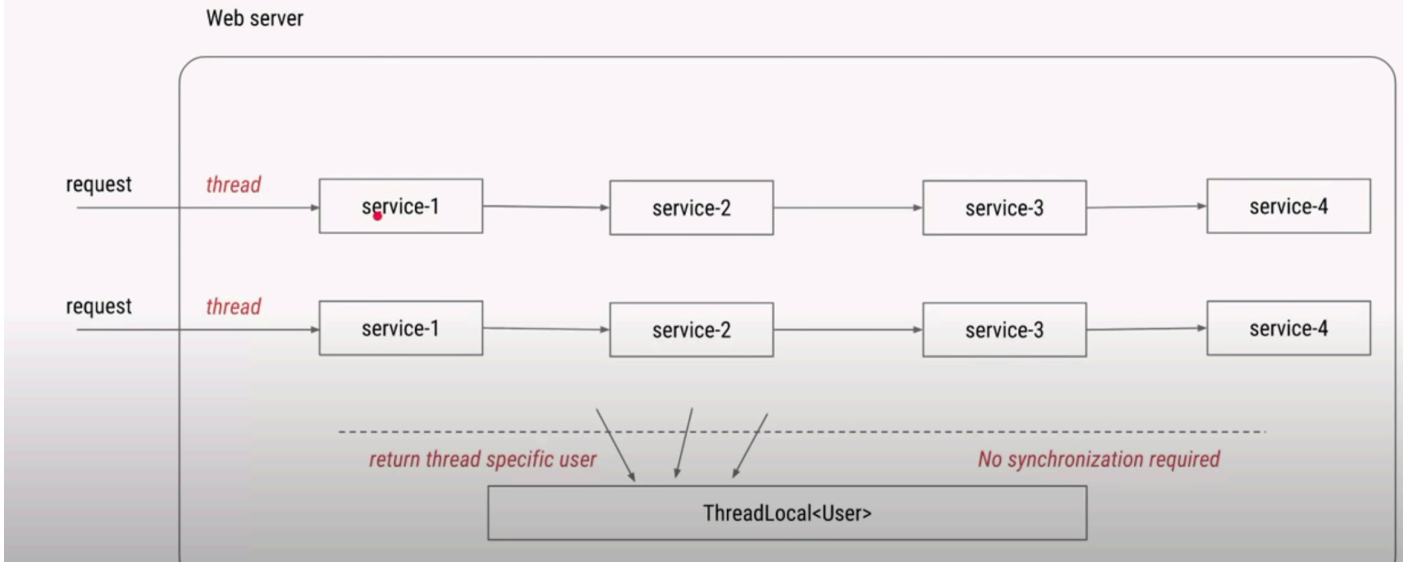
usecase 2:







Synchronization not required with ThreadLocal



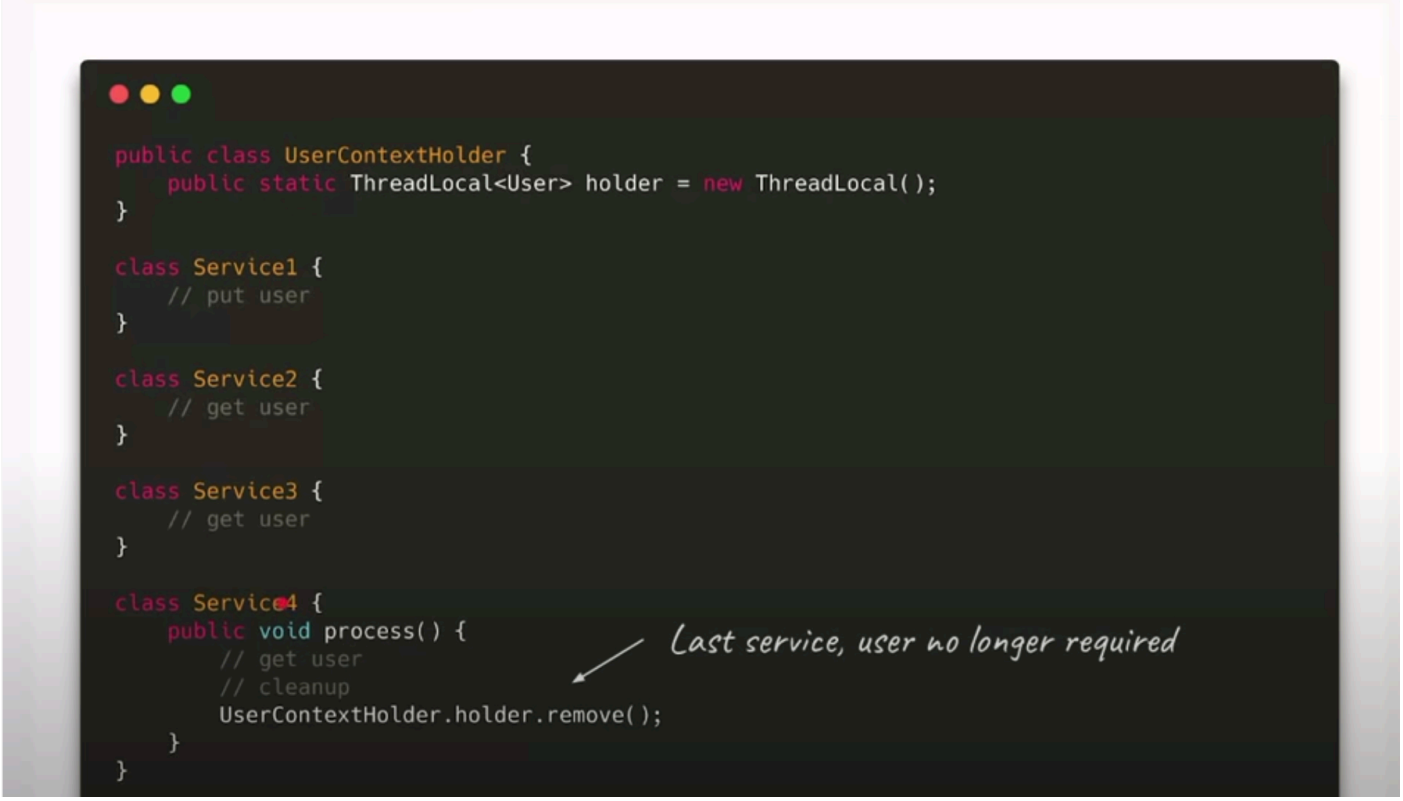

```
public class UserContextHolder {
    public static ThreadLocal<User> holder = new ThreadLocal();
}

class Service1 {
    public void process() {
        User user = getUser();
        UserContextHolder.holder.set(user);
    }
}

class Service2 {
    public void process() {
        User user = UserContextHolder.holder.get();
```

Set it for this thread

Get user for this thread

```
        // process user
    }
}
```

Important to cleanup once job is over

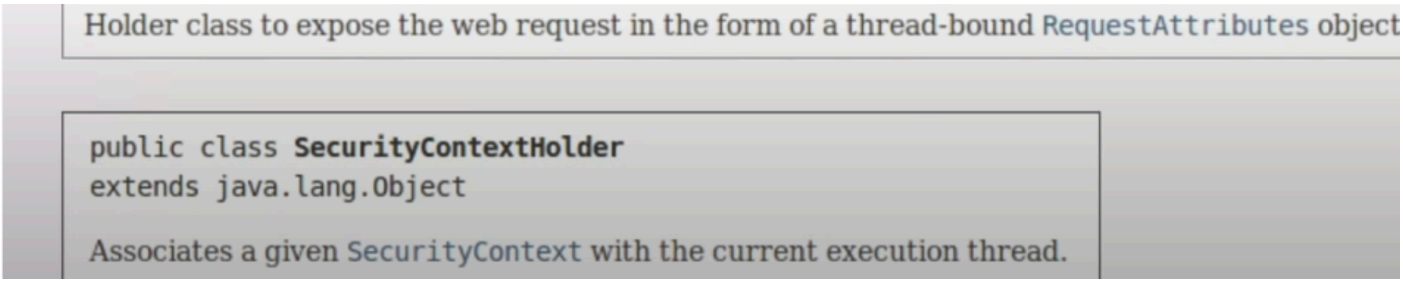


If we are using spring framework then try to use below context to avoid and handle threadlocal by ourself.

Spring framework uses lot of Context holders

- LocaleContextHolder
- TransactionContextHolder
- RequestContextHolder
- SecurityContextHolder
- DateTimeContextHolder

```
public abstract class RequestContextHolder
extends java.lang.Object
```



If we are using spring framework then try to use below context to avoid and handle threadlocal by ourself.