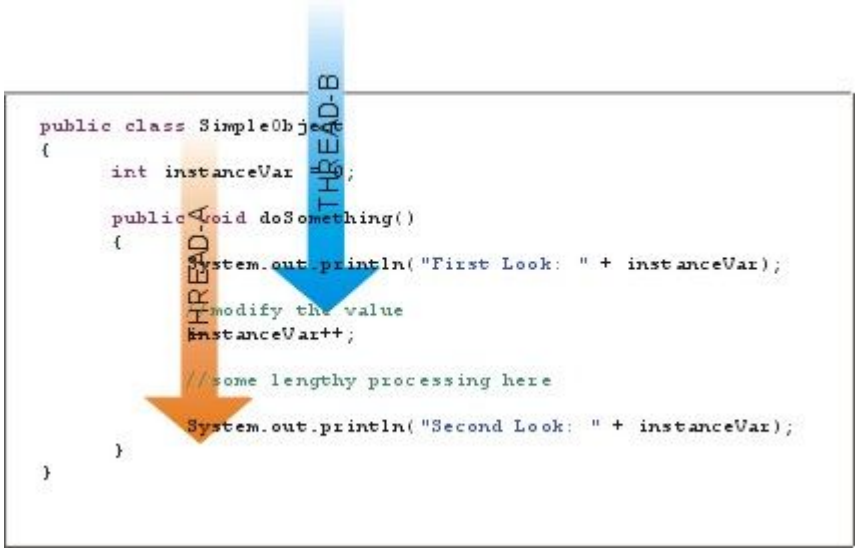
Threads and Synchronization

Keeping your data safe in a multithreaded world

Multithreading

Threads

- A thread is a single execution process.
- An individual, sequential flow of control within a program
- Memory is shared between threads.



Sharing Memory

- Since threads share memory (objects, fields, etc) you can run into situations where data corruption can occur.

  Example: Take a look again at the previous figure.

Deadlocks

- A situation where Thread A with Resource A is waiting on Thread B to give up Resource B, but Thread B is waiting on Thread A to give up Resource A. This is known as a deadlock and generally results in an unresponsive program or heap overflow.

Back to Servlets

- Much of the servlet lifecycle is handled by our application container (ie Tomcat).
- It is responsible for creation, destruction, and execution of servlet instances.
- It ensures that servlets are handled efficiently and to do this multithreading is needed.

Consequences of Servlets

- The application server creates only a single instance of a servlet, however it could potentially attach multiple threads to it.
- Because of this any instance fields can be potentially read and written to by multiple threads at any arbitrary time.
- Meaning our instance fields ARE NOT thread safe!

Best Defense: Avoidance

- The best approach towards thread safety is to avoid it all together.
- Use as few instance fields within a servlet as possible. If a field can be refactored into a local variable then do it.

Secondary Defense: Partial synchronization

- Use the *synchronized* keyword to define a block of code that can only be accessed by one thread at a time. Other threads will have to wait.
- Only synchronized as little code as possible because synchronized blocks can easily become bottlenecks for your application.

Synchronized Methods

Adding the synchronized keyword in the method header makes it so that one and only one thread can call the method. All other threads must wait.

Note that constructors cannot be synchronized.

Reentrant Synchronization

A thread cannot obtain a lock owned by another thread. However, it can obtain a lock that it currently owns. This is known as reentrant synchronization and must be avoided.

Last Defense: Full Synchronization

- To tell the servlet container that a particular servlet should be single threaded, implement the *SingleThreadModel* interface. This is used as a tag by the container to know which servlets are single threaded.
- Naturally, the easiest approach is also the most expensive.

Additional Java Thread Safety Data Structures

Concurrent Collections

ConcurrentMap

This is a sub interface of the *Map* interface and provides thread safety guarantees.

Implementing Classes: ConcurrentHashMap

Blocking Queues

- When a thread ‘blocks’ it stop execution waiting on some event to occur.
- Blocking queues are queues that will block when…
  - The queue has run out of space. Block until space is available.
  - The queue is empty. Block until it becomes non-empty.

Atomic Variables

A set of classes that support atomic operations on single variables using standard *get* and *set* methods.

AtomicBoolean, AtomicInteger, AtomicIntegerArray, etc.

Concurrent Randomness

ThreadLocalRandom is used for thread local random numbers. It results in less contention then *Math.random* and results in better performance.

Revisiting Deadlocks

There is no data structure or keyword to avoid deadlocks. Be very careful when using calls that block such as BlockingQueues.