Happens-before

Current-day servers and desktop computers come equipped with multicore CPUs.

These processors facilitate concurrency in application programming, which promotes multithreading, allowing for higher efficiency and productivity within our applications.

Single-core CPUs also employ multithreading, such as in supporting multiuser access. Such access requires the use of **concurrency** — two tasks advancing on overlapping time periods.

Unlike processes that each come with their own set of data, threads share the data amongst themselves, providing efficiency to the program due to threads needing less overhead management.

Sharing variables between threads and not requiring external communication between them makes it easier for multi-threaded applications.

With this advance, we now have to consider the different varieties of computer CPU architectures provided by various manufacturers. These can vary in register, memory, and caching schemes, which then becomes an issue with language developers due to possible nonstandard access to application data. While not a big problem with single-threaded applications, it becomes a major concern with multithreading programs.

The problem arises when variables and objects are modified by one thread and accessed by additional threads. For the reasons of architecture mentioned above, the consistency of the data values may not be in a stable state — the way variables are seen by one thread may be different from what another thread sees. Similarly, the state of an object may seem different to other threads.

This topic will explain a set of rules that the Java language enforces to handle this significant issue.
These rules are collectively known by the term **happens-before**.

Shared data is not synchronized

Synchronization does not happen automatically. There are various reasons for this.

A memory model controls the conditions in which different threads see the variables of a program.

There are different levels of memory models.
The memory model with minimum guarantees is the processor architecture on which the JVM runs. This model is at the bottom of the stack and hands over the responsibility to provide thread safety to the operating system, the compiler, and the JRE.

The memory model explicitly provides special instructions for the higher components to enhance the shared memory guarantees required for multithreading activities.
These special instructions provided by the hardware memory model are called memory barriers and will be employed by the **Java Memory Model** – **JMM**.

The JMM works with the processor architecture memory model and guarantees memory variable visibility. **The Java Virtual Machine — JVM —** is responsible for dealing with any differences between the JMM and the hardware architecture.
This is done by the use of the memory barriers mentioned earlier. Bringing all this together, on the one hand, we need the hardware to have the maximum freedom to be as efficient as possible, and on the other, the JVM must perform on any number of processor architectures.

At this point, we have an orchestrated scheme to try and manipulate variables and code objects in multithreaded shared data, expecting to obtain sequential order as prescribed by the program code. This also allows the language to fulfill its promise of platform independence and portability.

So far, these memory models don't provide the sequential consistency needed to write concurrent multithreading programs.
The main reasons that synchronization of multiple threads is not possible at this point in this topic can be coined into one term called reordering. **Reordering** is a group of reasons that cause incorrect results in multithreaded programs. Events that fall into this category are delayed operations, out-of-order operations, and race conditions.
It is also possible that the JMM will permit reordering issues since it is not designed to catch all synchronization requirements.
Reordering occurs at the different memory model levels and is the cause of unexpected program behavior. Compilers and runtime engines can also exercise a reordering to optimize program code.
These actions could force the JMM not to honor its visibility guarantees. To overcome all these drawbacks we have encountered so far, let's spell out the solution to all these multithreading issues. The key is the concept of **synchronization**.

Synchronization prevents reordering operations that impede JMM visibility guarantees.

The term **action** is used to describe what the JMM does within threads. JMM actions include variable read and write, monitor locks and unlock, as well as thread start and join operations.

Since we are working with a procedural language, our code instructions must be run in order.

To uphold its guarantee, partial ordering is defined within the JMM. For example, if we have actions A1, A2, A3, and A4, a partial ordering is A1 happens-before A2, and then A2 happens-before A3.

Action A4 is not in the ordering and could happen before or after as it was not selected as critical, and its order does not impact other operations. This partial ordering is called **happens-before**.

You can think about happens-before as a set of rules defined by the JMM that must be applied to achieve a correctly synchronized program.

Program order is the first happens-before rule that enforces order within a single thread. For instance, Thread T1 could be executing actions A1 and A2.

There is a guarantee that T1 runs A1 and A2 as if they were executed in program order.

This example is executed in a single thread which could experience reordering, the same as would occur with multiple threads.

The program order rule guarantees that each action happens-before all subsequent actions on a single thread or multiple threads.

Volatile keyword

Various language mechanisms employ happens-before. One of these is the use of the keyword volatile. volatile guarantees coherent access to a field by multiple threads with visibility as its only guarantee.

A shared variable declared volatile can only be used with an assignment operation. This makes this mechanism a weaker synchronization tool due to it not using any locks.

Access to a volatile variable is guaranteed to be performed by the program order. The rule for volatile concerning a field is that a write happens-before every subsequent read.

There are other ways to provide happens-before. But first, let's look at an

example program that exhibits sequential consistency by using the volatile keyword. Let's look at a single-pump gas station. The pump updates the tank for every gallon dispensed to a customer. The pump and tank are run in separate threads with the volatile variable tankFuel, running in the main thread. If we remove the volatile keyword as seen below, we no longer have a visibility guarantee.

```
public class GasStation {

    public static int tankFuel = 10; // volatile keyword is required

    public static void main(String[] args) {

    }
}
```

The variable tankFuel is where the volatile keyword needs to be used in order to solve the visibility issue. Now we have a visibility guarantee assured by the promise of the happens-before rule for the volatile keyword.

```
1   public class GasStation {
2
3       public static volatile int tankFuel = 10;  // working program due to volatile
4       //public static int tankFuel = 10;         // broken program due to no volatile
5
6       public static void main(String[] args) {
7
8           new GasTank().start();
9           new GasPump().start();
10      }
11  }
```

In the class GasTank, the modification is carried out by the following:

```
GasStation.tankFuel--;
```

The thread running class GasTank is the only thread making modifications to the variable tankFuel.

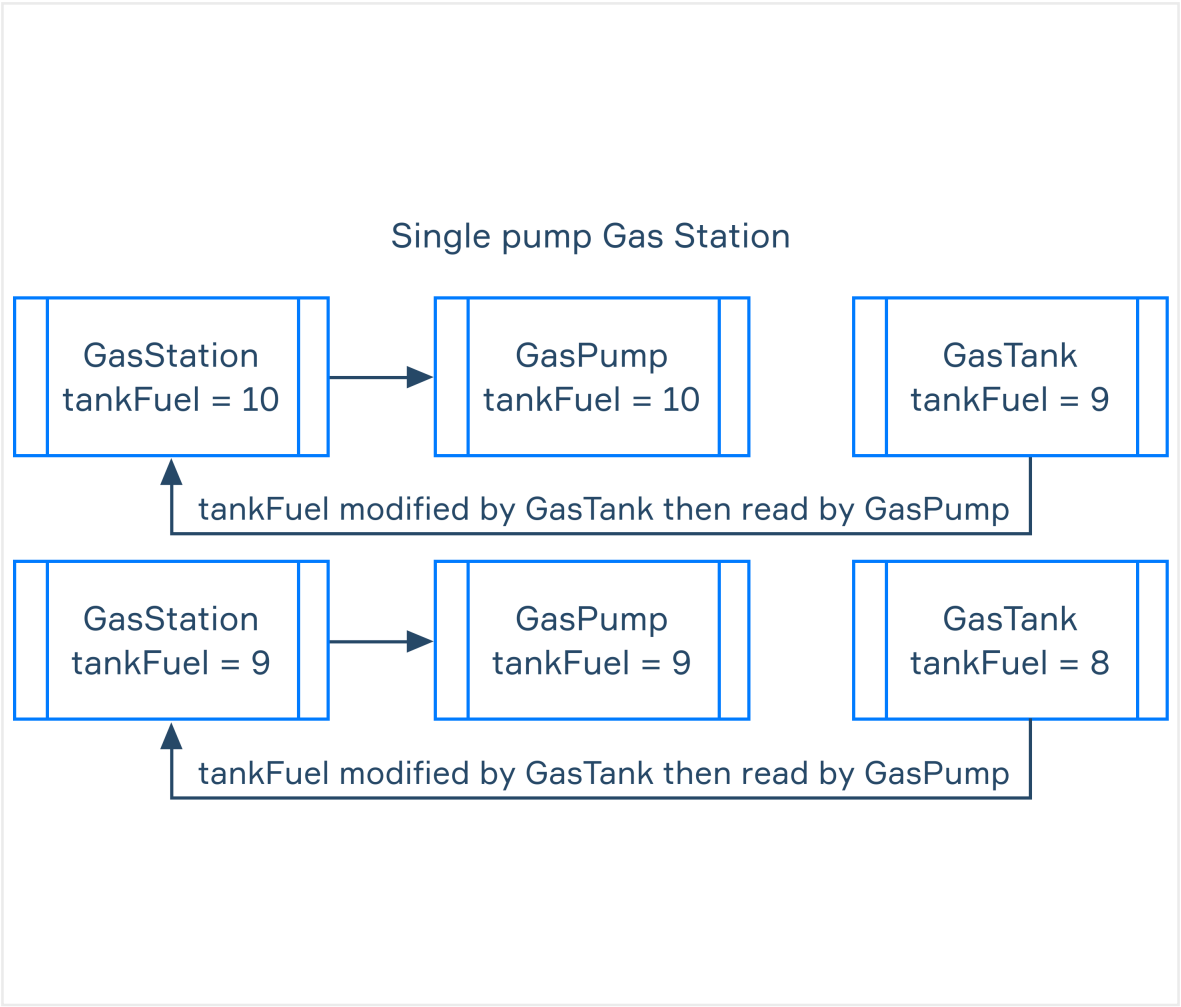
```
1   public class GasTank extends Thread {
2
3       @Override
4       public void run() {
5
6           while (GasStation.tankFuel > 0) {
7               System.out.println("Tank available fuel: " + GasStation.tankFuel);
8               GasStation.tankFuel--;
9
10              try {
11                  Thread.sleep(10);
12              }
13              catch (InterruptedException e) {
14                  e.printStackTrace();
```

```
15            }
16         }
17      }
18   }
```

The class GasPump below only performs reads on the volatile tankFuel variable. It will always reflect the up-to-date value.

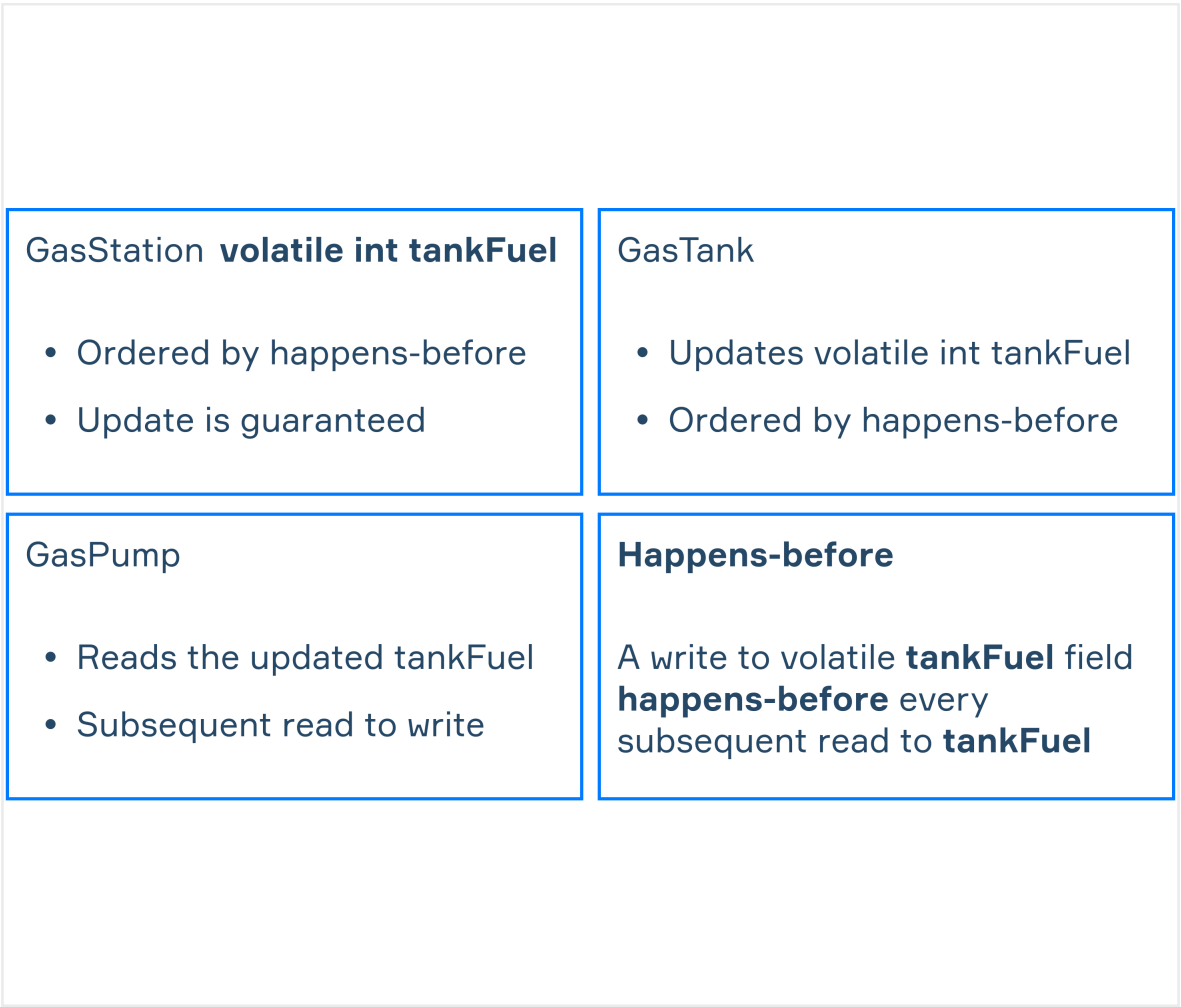
```
1    public class GasPump extends Thread {
2
3       @Override
4       public void run() {
5          int fuel = GasStation.tankFuel;
6          while (fuel > 0) {
7             if (fuel != GasStation.tankFuel) {
8                System.out.println("Pump available fuel: "
9                                  + GasStation.tankFuel);
10                fuel = GasStation.tankFuel;
11            }
12         }
13      }
14   }
```

The diagram below represents how the variable tankFuel is safely written and subsequently read. As the program runs, the tankFuel will correctly reflect its modified value.



This single-pump gas station benefited from using the volatile keyword due to its low synchronization requirements (modifying a single variable).

The partial ordering or the happens-before rule defined by the JMM was used by the volatile keyword. In effect, this guarded the read/write actions, which protected the tankFuel variable from corruption.



Synchronized keyword

The Java keyword synchronized is another mechanism of the language that provides exclusive locking of object methods. Every Java object has an intrinsic lock that protects a method when the synchronized keyword is used.

This prevents changes to mutable variables by multiple threads in an incorrect order. By using this synchronization mechanism, we are employing happens-before.

This, in turn, provides the program with the order required for accurate sequential operations in concurrent programs.

By simply using this synchronization mechanism, we are provided the safety for program variables and objects that one would expect when writing a piece of program code. synchronized guarantees both **visibility** and **atomicity**.

The next example switches to a more complex mechanism to guarantee a correctly synchronized program. Let's introduce a new gas station compared to the simple single-pump previously explored.

This gas station will be equipped with multiple pumps each running in separate threads creating a situation were variable modifications will arrive from more than one thread. It is now necessary to find a better synchronization solution. The synchronized keyword will be our choice in the following program.

As you can see, each pump will have access to a single tank for the entire gas station. Only two pumps are active but more could be added.

```
1   public class GasStation {
2
3       public static void main(String[] args) {
4
5           Tank tank = new Tank(100);
6           Pump pump1 = new Pump(tank);
7           Pump pump2 = new Pump(tank);
8
9           Runnable fillup1 = () -> pump1.pumpFuel(10);
10          Runnable fillup2 = () -> pump2.pumpFuel(10);
11
12          new Thread(fillup1,"car").start();
13          new Thread(fillup2, "truck").start();
14      }
15  }
```

Unlike the volatile keyword we previously used, this program uses the synchronized keyword to protect the use of the method setFuel(). This method modifies the variable fuel in the class Tank. The synchronized keyword will employ two happens-before rules to perform its duty: program order and monitor lock. Only one thread will have access to shared resources anytime.

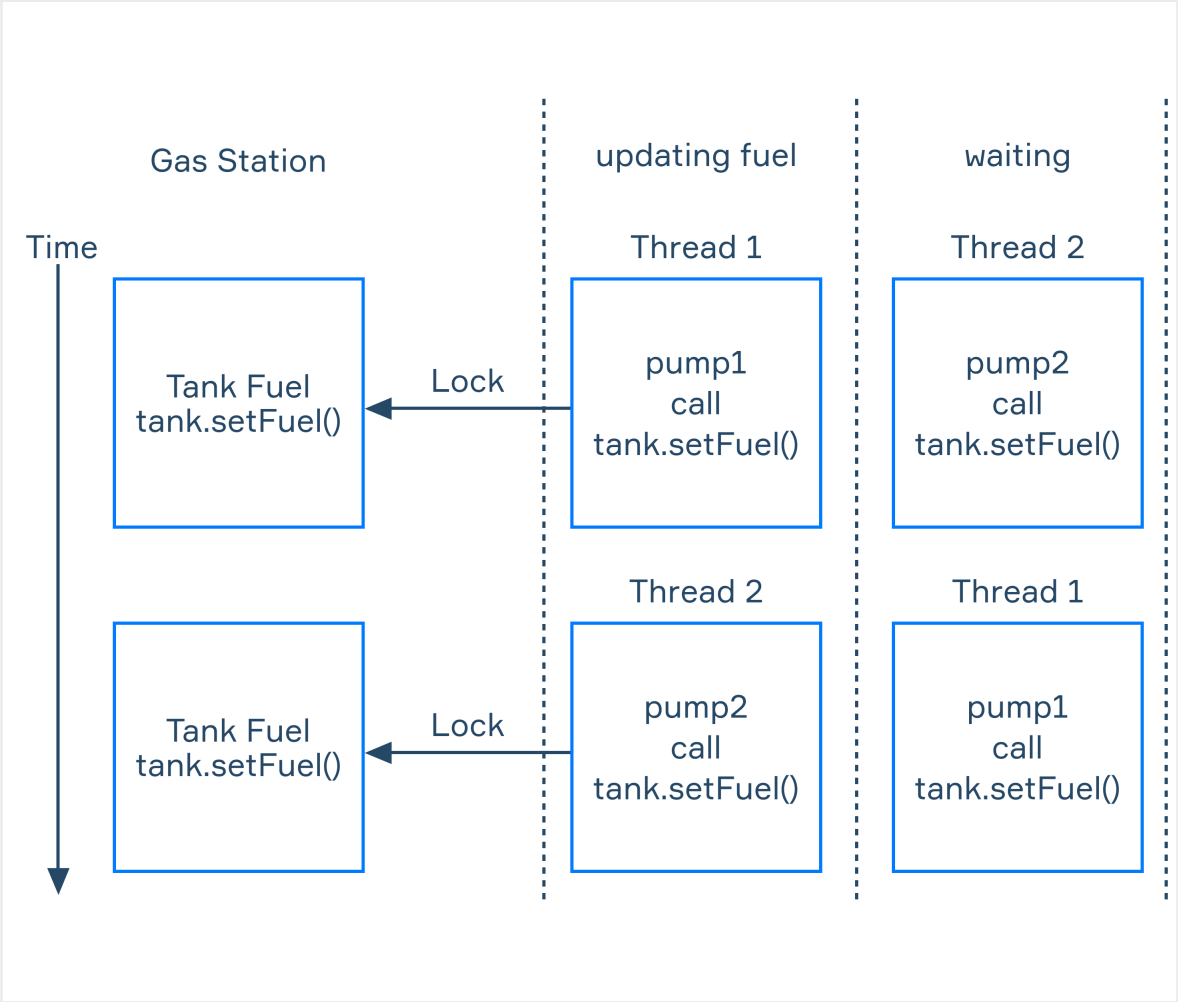
```
1   public class Tank {
2
3       private int fuel;
4
5       public Tank(int fuel) {
6           this.fuel = fuel;
7           System.out.printf("Fuel remaining %d\n",this.fuel);
8       }
9
10      public synchronized void setFuel() {
11          String name = Thread.currentThread().getName();
12
13          try {
14              Thread.sleep(10);
15          } catch (InterruptedException e) {
16              throw new RuntimeException(e);
17          }
18
19          this.fuel--;
```

```
20          System.out.printf("Filling up %5s, fuel remaining: %1d\n",
21                                          name, this.fuel);
22      }
23  }
```

In the Pump class, the method pumpFuel() calls the synchronized method tank.SetFuel. Any gas station pump that calls this method will be forced to respect the happens-before rules employed by the synchronized keyword.

```
1   public class Pump {
2
3       private final Tank tank;
4
5       public Pump(Tank tank) {
6           this.tank = tank;
7       }
8
9       public void pumpFuel(long fuel) {
10          while (fuel > 0) {
11              tank.setFuel();
12              fuel--;
13          }
14      }
15  }
```

The diagram below represents the Gas Station time frame as the different pumps take turns updating the tank fuel variable. The threads participate in mutual exclusion as each thread takes its turn accessing the synchronized method setFuel.



Monitors

A monitor in Java is a part of the synchronization scheme used in the language. It is a synonym of the term **intrinsic lock**. Monitors are available for every object since they are built into each object. Monitors enforce exclusive access to the state and visibility of an object. The **monitor-lock** rule spells out a happens-before relationship between a lock, unlock, and a subsequent lock on an object. An unlock on a previous monitor happens-before every subsequent lock on that same monitor. A thread can lock or unlock an objects monitor.

Object initialization

Object initialization also needs to have a happens-before relationship that protects them against object inconsistencies.
When a reference is made within one thread to an object just created, the term **publishing** describes the act of making this reference visible to other threads.

This object is now available to use with its reference variable. Objects can be published only when completely constructed. This will ensure a stable object.

The new object contains fields that could be written to, and the happens-before guarantees provided by the JMM are employed to provide safe publication, preventing stale fields in the new object.

Also, happens-before guarantees permit reference variable assignment to have the correct value. Since reordering is always a possibility, guarantees are only there if proper synchronization is exercised.

Thread start and join

When you start a Java program, the main thread is first to run automatically. Within this main thread, we can create additional "child" threads. Once its task is completed, the newly created thread will terminate and, subsequently, the main thread will finish.

This is a very simple example of what occurs in a multi-threading program and generally, more complex scenarios take place.

Starting a thread requires the use of the Thread class. The Thread class has several methods to manage all created threads running at any time in the life of a program.

Creating a new thread is done using the start method creating a new thread object. By invoking start in a statement, we have a happens-before relationship between the invoking statement with all its previous statements to every statement executed by the new thread.

Any changes made to variables before the statement containing start will be visible to all statements in the new thread. When the the new thread has completed its task, the Thread class method join is commonly called, and a return back to the main thread occurs.

After calling join, the thread stops its execution. All the statements previously executed by the now-terminated thread have a happens-before relationship to any statements that would be subsequently executed in the main thread.

Conclusion

Different CPUs have nonstandard memory models that are optimized for performance. This focus on performance puts the program code in jeopardy of producing incorrect results. Reordering is the term used to group the major causes harming the program execution due to incorrect program order. Synchronization mechanisms are provided and built into the language to compensate for these less clear traps. A set of rules called happens-before is defined by the JMM to solve issues encountered in multi-threaded applications. These rules must be applied whenever multithreaded synchronization issues could occur. This consideration will produce reliable results guaranteeing the

accuracy required by a program.