Singleton Pattern

Contents

- Usage of Singleton pattern
- Singleton pattern implementation for single-thread
- Singleton pattern implementation for multi-thread
- Related patterns

The Singleton Pattern

- **Purpose**
  - Ensures that only one instance of a class is allowed within a system.
- **Use When**
  - Exactly one instance of a class is required.
  - Controlled access to a single object is necessary.

One of a Kind

- What kind of a knight are you?

One of a Kind Objects

- One and only one
  - Window Manager, Printer Spooler, Thread Pool Manager, Caches, Logging, Factory
  - We want to instantiate them only once
- Why is it difficult after all?
  - How about global variables?
  - Multi-threading issue

Usage of Singleton

```
public class AnyClientProgram {
    // Singleton s = new Singleton();
    // the above triggers compilation error
    Singleton s = Singleton.getInstance();
}
```

Clients are not allowed to use new operator!

The Skeleton of Singleton

```
public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables

    private Singleton() {}

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

    // other useful methods
}
```

Beware of Concurrency!!

Design Points

- Make the constructor be private
  - `private Singleton() {}`
  - Private constructor ?
  - Otherwise …
- Provide a getInstance() method
  - `public static Singleton getInstance()`
  - Should it be static?
  - Why public ?
- Remember the instance once you have created it
  - `private static Singleton uniqueInstance`
  - ```
    if (uniqueInstance == null)
      uniqueInstance = new Singleton();
    ```
    ```

Exercise: Applying Singleton

```
public class ChocolateBoiler {
   private boolean empty;
   private boolean boiled;
   
   
   ChocolateBoiler() {
         empty = true;
         boiled = false;
   }
   
   
   public void fill() {
       if(isEmpty()) {
           empty = false;
           boiled = false;
           // fill the boiler with a milk/chocolate mixture
       }
   }
   // rest of ChocolateBoiler code...
}
```

Solution

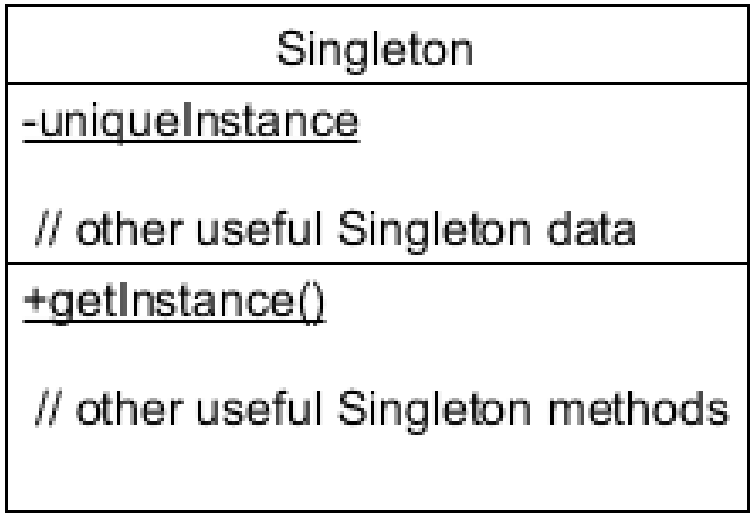
```
public class ChocolateBoiler {
  private boolean empty;
  private boolean boiled;
  private static ChocolateBoiler uniqueInstance;

  private ChocolateBoiler() {
         empty = true;
         boiled = false;
  }

  public static ChocolateBoiler getInstance() {
         if (uniqueInstance == null) {
                uniqueInstance = new ChocolateBoiler();
         }
         return uniqueInstance;
  }
       ..

}
```

Class Diagram



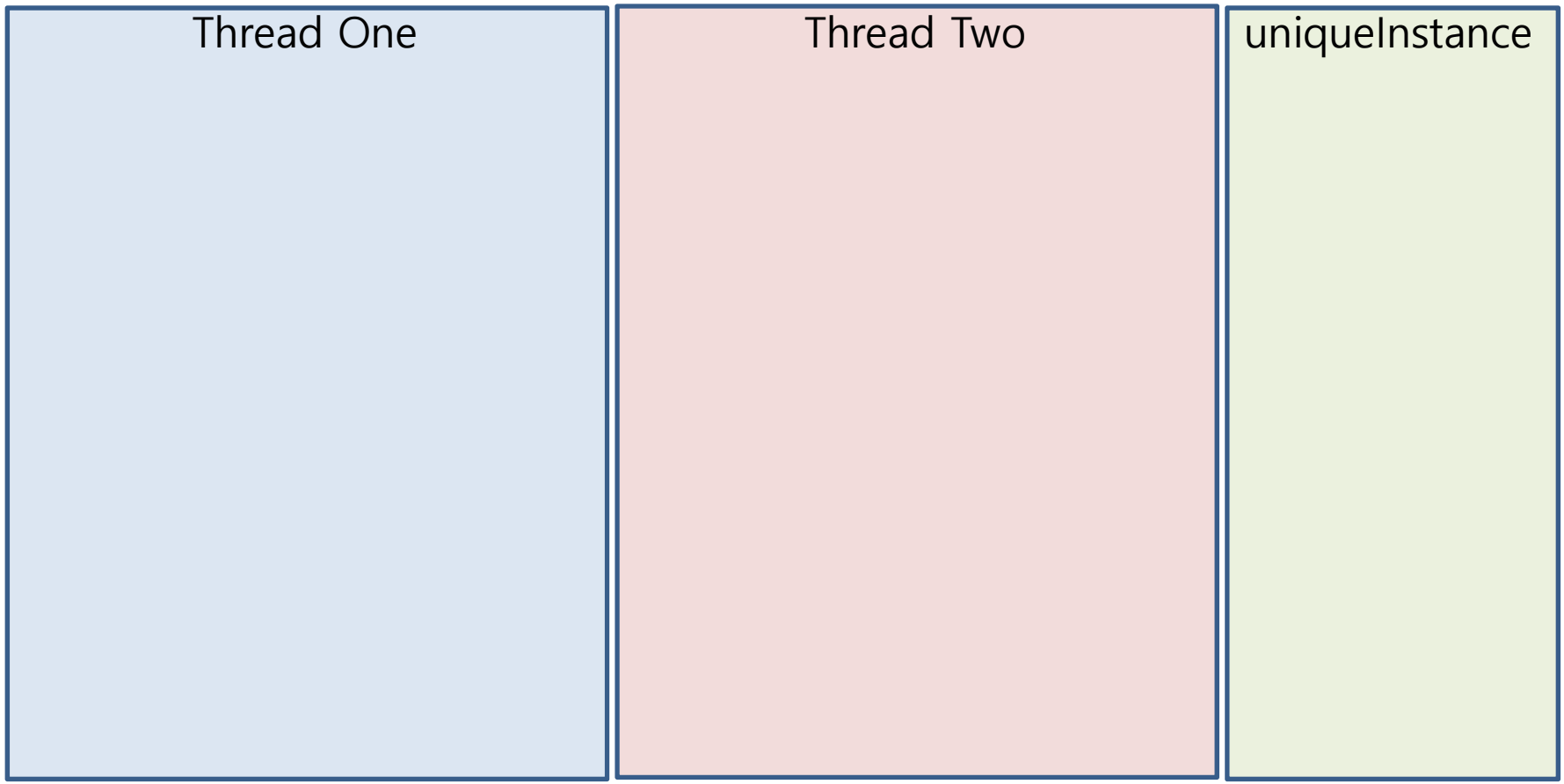
Using Singleton on Multi-threads

- Thread One and Thread Two executes

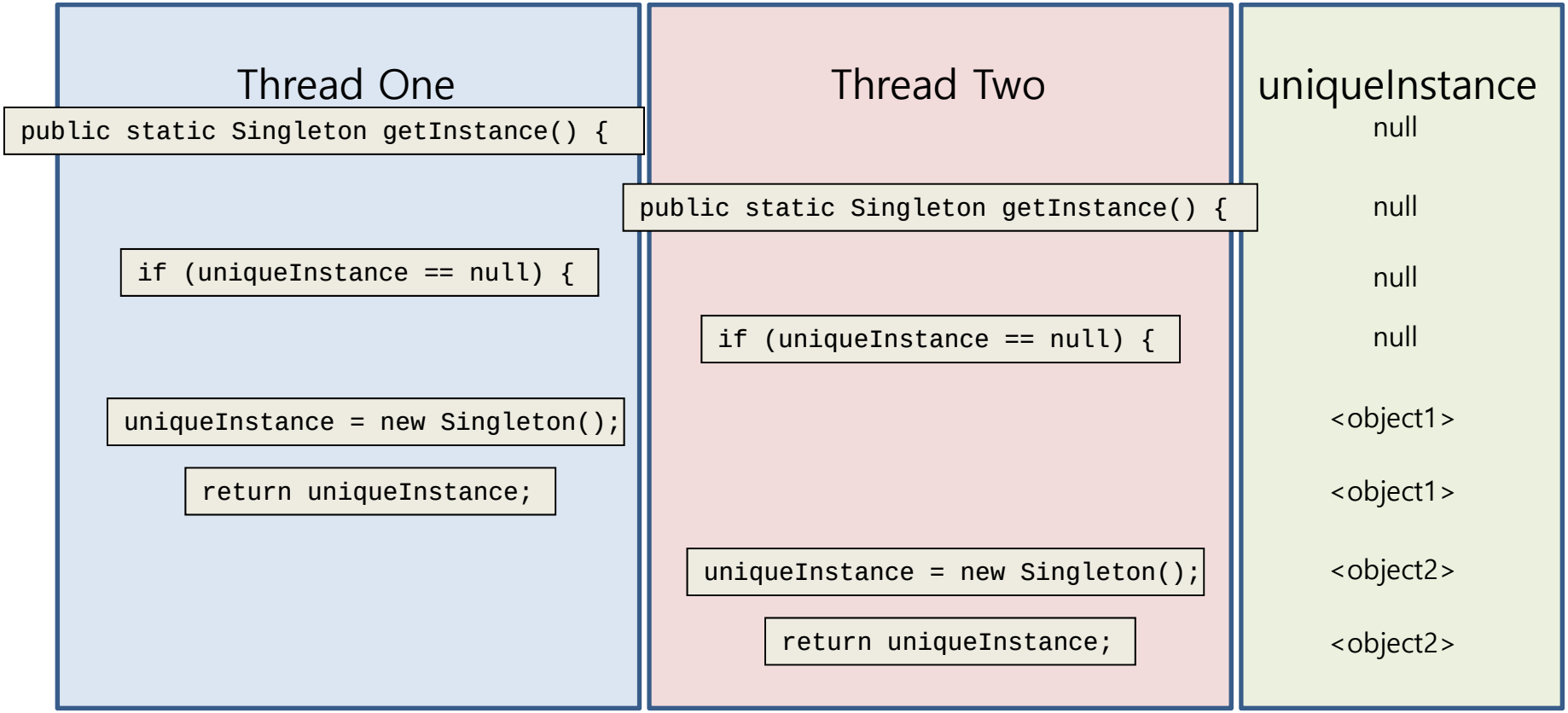
```
ChocolateBoiler boiler = ChocolateBoiler.getInstance();
boiler.fill();
boiler.boil();
boiler.drain();
```

What happened?

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



What happened?




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

Solving the Problem (Option 1)

```
public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables

    private Singleton() {}

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

    // other useful methods
}
```

Isn't this too much locking?

Solving the Problem (Option2)

```
public class Singleton {
    private static Singleton uniqueInstance = new Singleton();

    // other useful instance variables

    private Singleton() {}

    public static Singleton getInstance() {
        return uniqueInstance;
    }

    // other useful methods
}
```

Remind the overhead of global variables

Let's go back to the solution for the single thread

```
public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables

    private Singleton() {}

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

    // other useful methods
}
```

Is this correct?

```
public class Singleton {
    private static Singleton uniqueInstance = null;

    // other useful instance variables

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            synchronized(Singleton.class) {
                uniqueInstance = new Singleton();
            }
        }
        return uniqueInstance;
    }

    // other useful methods
}
```

Developing Idea

1. Check that the variable is initialized (without obtaining the lock). If it is initialized, return it immediately.

2. Obtain the lock.

3. **Double-check** whether the variable has already been initialized: if another thread acquired the lock first, it may have already done the initialization. If so, return the initialized variable.

4. Otherwise, initialize and return the variable.

Then, how about this?

```
public class Singleton {
    private static Singleton uniqueInstance = null;

    // other useful instance variables

    private Singleton() {}

        public static Singleton getInstance() {
            if (uniqueInstance == null) {
                synchronized(Singleton.class) {
                    if (uniqueInstance == null)
                        uniqueInstance = new Singleton();
                }
            }
            return uniqueInstance;
    }
    // other useful methods
}
```

Double-Checked Locking

At first glance, the previous algorithm seems like an efficient solution to the problem. However, this technique has many subtle problems and should usually be avoided.

In a scenario, thread *A* notices that the value is not initialized, so it obtains the lock and begins to initialize the value.

Due to the semantics of some programming languages, the code generated by the compiler is allowed to update the shared variable to point to a partially constructed object before *A* has finished performing the initialization.

Thread *B* notices that the shared variable has been initialized (or so it appears), and returns its value. Because thread *B* believes the value is already initialized, it does not acquire the lock. **If the variable is used before *A* finishes initializing it, the program will likely crash.**

Double-Checked Locking

One of the dangers of using double-checked locking in J2SE 1.4 and earlier versions is that it will often appear to work.

Depending on the compiler, the interleaving of threads by the scheduler and the nature of other concurrent system activities, failures resulting from an incorrect implementation of double-checked locking may only occur intermittently.

Reproducing the failures can be difficult.

Corrected Double-Checked Locking (Option 3)

```
public class Singleton {
    private volatile static Singleton uniqueInstance = null;
    // other useful instance variables

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            synchronized(Singleton.class) {
                if (uniqueInstance == null)
                    uniqueInstance = new Singleton();
            }
        }
        return uniqueInstance;
    }
    // other useful methods
}
```

Using volatile keyword

Thread *A* notices that the value is not initialized, so it obtains the lock a
nd begins to initialize the value.
~~Due to the semantics of some programming languages, the code gener~~
~~ated by the compiler is allowed to update the shared variable to point t~~
~~o a partially constructed object before *A* has finished performing the ini~~
~~tialization.~~
Only after complete construction of the object, the shared variable is se
t to point to the object.
Thread *B* notices that the shared variable has been initialized, and retur
ns its value. Because thread *B* believes the value is already initialized, it
does not acquire the lock.
Using the value in thread B does not trigger any problem since the obje
ct pointed by the value is completely constructed.

More on Volatile Variables

```
program Test {
    global boolean flag;
    public void foo() {
        flag = false;
        if (flag) {
           // this could happen
        }
    }
}
```

- History of Volatile Variables
  - The volatile keyword was introduced to the language as a way around optimizing compilers.
  - An optimizing compiler might decide that the body of the if statement would never execute, and not even compile the code.
  - Declaring variables with the volatile keyword tells the compiler not to optimize out sections of code by predicting the value of the variable at compile time.

What does volatile mean?

- C/C++ spec
  - There is no implementation independent meaning of volatile
- Situation a little better with Java Technology
  - volatile reads/writes guaranteed to go directly to main memory
  - e.g. can’t be cached in registers or local memory

Working Memory v.s. Main Memory

- Every thread has a ***working memory*** in which it keeps its own ***working copy*** of variables that it must use or assign. As the thread executes a program, it operates on these working copies.
- The main memory contains the ***master copy*** of every variable.

Synchronization

- Atomicity
  - locking to obtain mutual exclusion
  - Atomic read/write granularity
- Visibility
  - ensuring that changes in object fields on one object are seen in another thread
- Ordering
  - ensuring that you aren’t surprised by the order in which statements are executed

Will this print out “x < y” ?

```
class Something {
  private int x = 0;
  private int y = 0;
  public void write() {
    x = 100;
    y = 50;
  }
  public void read() {
    if (x < y)
      System.out.println("x < y");
  }
}
```

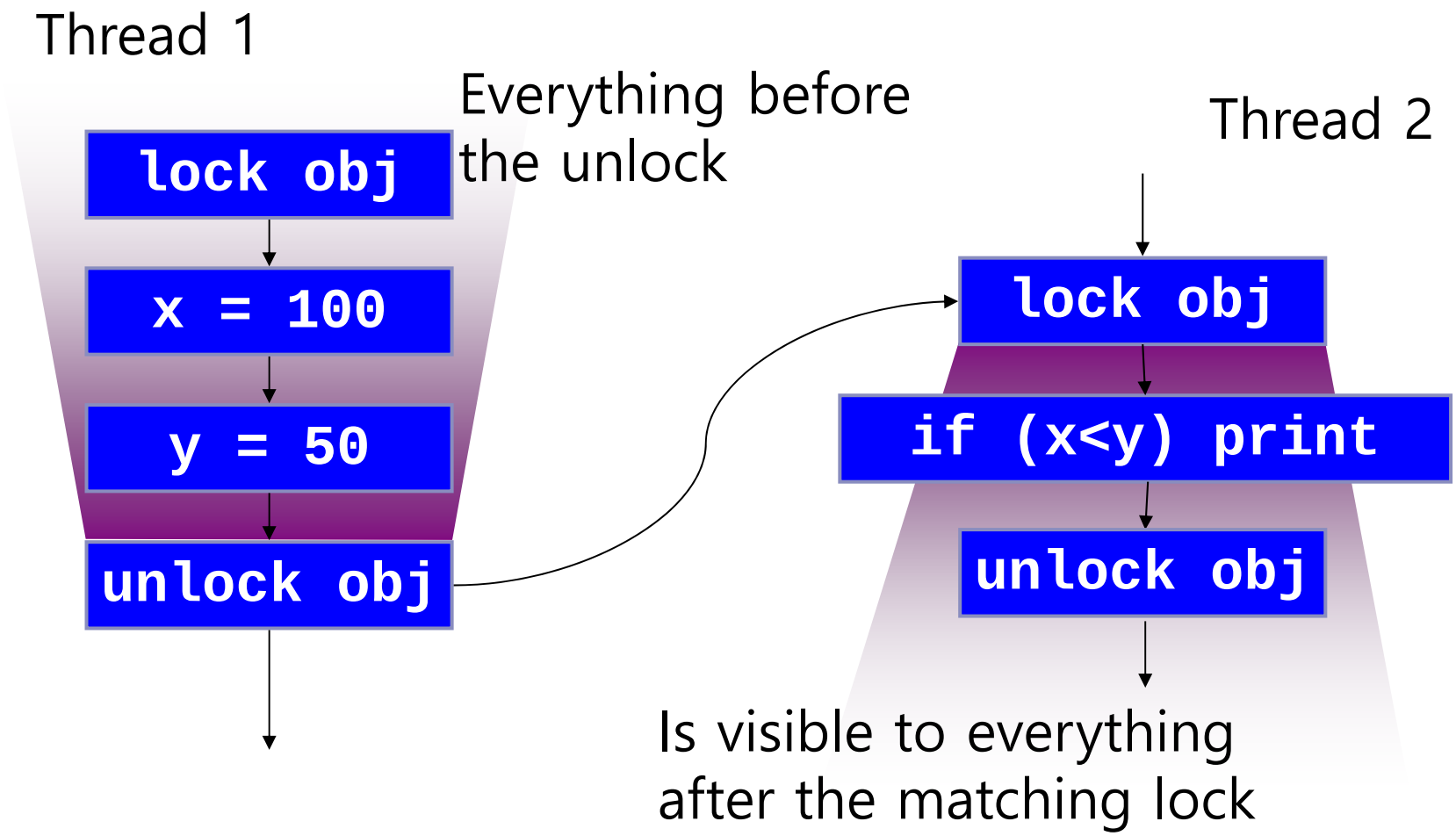
```
public class Main {
  public static void main(String[] args) {
    Something obj = new Something();
    new Thread() {
      public void run() {
        obj.write();
      }
    }.start();
    new Thread() {
      public void run() {
        obj.read();
      }
    }.start();
  }
}
```

This NEVER prints out “x < y”

```
class Something {
  private int x = 0;
  private int y = 0;
  public void synchronized write() {
    x = 100;
    y = 50;
  }
  public void synchronized read() {
    if (x < y)
      System.out.println("x < y");
  }
}
```

```
public class Main {
  public static void main(String[] args) {
    Something obj = new Something();
    new Thread() {
      public void run() {
        obj.write();
      }
    }.start();
    new Thread() {
      public void run() {
        obj.read();
      }
    }.start();
  }
}
```

When are actions visible and ordered with other Threads?



Volatile in Java

- In all versions of Java
  - There is a **global ordering** on the **reads and writes** to a **volatile** variable.
  - This implies that every thread accessing a volatile field will **read its current value** before continuing, instead of (potentially) using a cached value.
  - However, there is no guarantee about the relative ordering of **volatile reads and writes** with **regular reads and writes**, meaning that it's generally not a useful threading construct.

- In Java 5 or later
  - Volatile reads and writes establish a happens-before relationship, much like acquiring and releasing a mutex.
  - Using volatile may be faster than a lock, but it will not work in some situations.

Will this work?

```
class Runner extends Thread {
    private boolean quit = false;
    public void run() {
        while (!quit) {
           // do something
        }
    }
    public void shutdown() {
        quit = true;
    }
}

public class Main {
    public static void main(String[] args) {
        Runner runner = new Runner();
        runner.start();
        // do some heavy jobs
        runner.shutdown();
    }
}
```

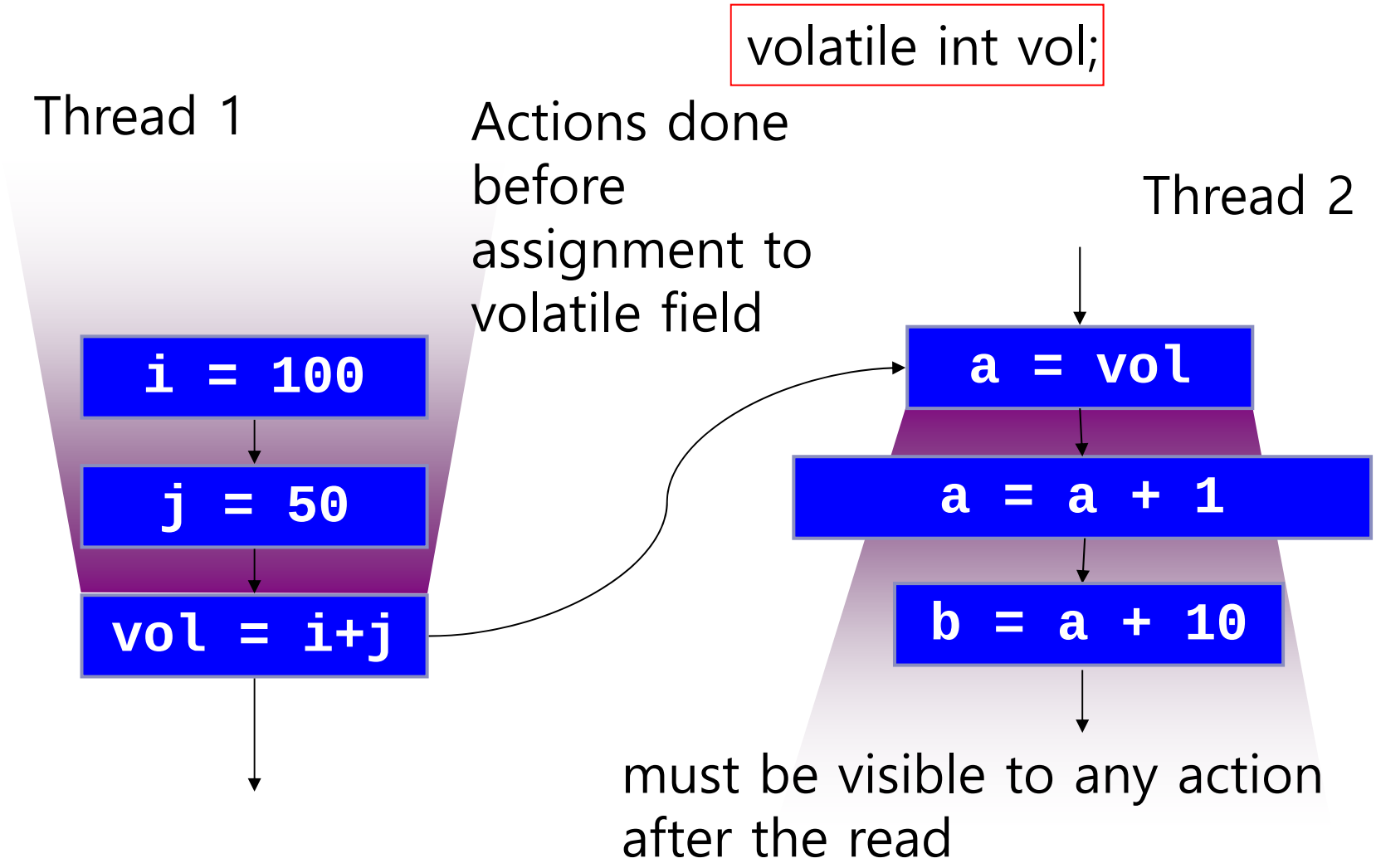
Using volatile (Corrected)

```
class Runner extends Thread {
    private volatile boolean quit = false;
    public void run() {
        while (!quit) {
            // do something
        }
    }
    public void shutdown() {
        quit = true;
    }
}

public class Main {
    public static void main(String[] args) {
        Runner runner = new Runner();
        runner.start();
        // do some heavy jobs
        runner.shutdown();
    }
}
```

Volatile is used to guarantee visibility of writes: quit ***must*** be declared volatile. Otherwise, compiler could keep the write in register

Visibility with volatile



Reviewing the Options

- Synchronize the getInstance() method (Option 1):
  - A straightforward technique that is guaranteed to work. It causes small impact on run-time performance due to frequent locking.
- Use eager instantiation (Option 2):
  - In case we are always going to instantiate the class, then statically initializing the instance would cause no concerns.
- Double checked locking (Option 3):
  - A perfect solution w.r.t performance. However, double-checked locking may be overkill in case we have no performance concerns. In addition, we'd have to ensure that we are running at least Java 5

Related Patterns and Summary

- Abstract Factory, Builder, and Prototype can use Singleton in their implementation.
- Facade objects are often Singletons because only one Facade object is required.
- State objects are often Singletons.

- Singleton pattern
  - creates at most one instance
  - Implementation: beware of multi-threaded issue