Safe Object Sharing

- **Shared read-only**
  - Initialization Guarantees
  - Immutability
- **Not shared**
  - Stack Confined: Method Local Variables
  - Thread Confined: ThreadLocals

If multiple threads access the same mutable state variable without appropriate synchronization, *your program is broken*. There are three ways to fix it:

- Use synchronization whenever accessing the state variable ✓
- Make the state variable immutable
- Don't share the state variable across threads

[JCIP 2]

Initialization-Safety-Guarantee

- **Problem: Clients may get a reference to an uninitialized object!**

```
final class Account {
   private int balance;
   public Account(int balance) { this.balance = balance; }
   public String toString() { return "" + balance; }
}
```

```
class Company {
   private Account account = null;
   public Account getAccount() { // lazy initialization
      if(account == null) account = new Account(10000);
      return account;
   }
}
```

```
T0: company.getAccount().toString();
```

```
T1: company.getAccount().toString();
```

Initialization-Safety-Guarantee

- **Fix: Use of volatile (safe publication)**

```
class Company {
   private volatile Account account = null;
   public Account getAccount() {
      if(account == null) {
         account = new Account(10000);
      }
      return account;
   }
}
```

  - Happens-before relation guarantees that fields set in the Integer-constructor are visible (as the invocation of constructor *happens-before* the assignment to the volatile field `account`)
  - Remark: no singleton guarantee (assumption: not required)

Initialization-Safety-Guarantee

- **Fix: Initialization-Safety-Guarantee**
  - JMM guarantees that final fields are only visible after they have been initialized!

```
class Account {
   private final int balance;
   public Account(int balance) { this.balance = balance; }
   public String toString() { return "" + balance; }
}
```

  - Guarantee: if a thread sees a reference to a Account instance, it has the guarantee to see the initialized final fields

Initialization-Safety-Guarantee

- **Guarantees of the JMM**
  1. Final fields (of primitive type and references) are visible after initialization
     - The initial (final) values are always visible, not the default (0, null, …) values
     - This guarantee only holds for the final fields!
  2. For final references the JMM guarantees that all referenced objects are visible after initialization if accessed over the final reference.

- **Consequences**
  – At the end of the initialization phase of an object with final fields, all final fields (and its transitive hull) are flushed into main memory
  – i.e. this data becomes visible BEFORE the address of the object becomes visible
  – Useful for immutable objects
  Advise: declare fields in immutables as final (for initialization guarantee)

Initialization-Safety-Guarantee

- **Example: Currencies Map**

```
class SafeCurrencies {
   private final Map<String, String> currencies;
   public SafeCurrencies () {
      currencies = new HashMap();
      currencies.put("United States", "USD");
      currencies.put("Germany", "EUR");
      currencies.put("Switzerland", "CHF");
      ...
      currencies.put("Zimbabwe", "ZWD");
   }
   public String getCurrency(String country){
      return currencies.get(country);
   }
}
```

  – As `currencies` is declared final, threads accessing `getCurrency` see at least the state of the map at the end of the constructor

Requirement: Safe construction

- **Initialization-Safety is only guaranteed if an object is accessed after it is fully constructed**
  - Do not allow the this reference to escape during construction
    - Don't assign `this` to a variable where other code can access it
      - e.g. a static variable
    - Don't register `this` as a listener in the constructor
    - Don't start threads in the constructor which act on `this`
    - Do not pass `this` to an *alien met*hod in the constructor
      - method in another class (i.e. not fully specified by current class)
      - overridable (non-private and non-final) method
  - Use a factory method if initialization requires multiple steps
    - in particular if the created object needs to be assigned to a static variable or
    - if the created object needs to be registered as a listener

Listener Registration in Constructor: BAD

```
public class ThisEscape {
    public final int i;

    public ThisEscape(Button source) {
        source.registerListener(new ClickListener() {
            // this escapes here
            public void buttonClicked() {
                ThisEscape.this.doSomething();
            }
        });
        i = 42;
    }

    public void doSomething() {
        System.out.println(i);
    }
}
```

Listener Registration in Factory Method: GOOD

```
public class ThisNotEscape {
    public final int i;
    private ThisNotEscape() { i = 42; }

    public static ThisNotEscape create(Button source) {
        final ThisNotEscape notEscape = new ThisNotEscape();
        source.registerListener(new ClickListener() {
            public void buttonClicked() {
                notEscape.doSomething();
            }
        });
        return notEscape;
    }
    ...
}
```

Final vs Volatile

- **final**
  - Only at the end of the constructor a (partial) flush happens
  - **Only the first access** leads to a (partial) refresh
  - After first access no refresh is performed (final fields cannot change)
  - Changes in referenced objects do ***not*** become visible automatically

- **volatile**
  - **Each read access** guarantees that the most recent data is seen
  - No guarantees for referenced objects (beyond happens-before guarantees)

Summary

- **Java Memory Model**
  - Requires the JVM to maintain only *within-thread as-if-serial semantics*
  - Defines happens-before relation across threads
    - Locking / Unlocking (using synchronized-blocks or java.util.concurrent.locks)
    - Volatile read / write
    - Thread start / observation of termination
  - Inter-Thread actions are visible only if they are ordered by happens-before relations

- **Initialization-Safety-Guarantee**
    - Values reachable through final fields are visible as of construction time
    - Only if object is properly constructed (no escaping)

Safe Object Sharing

- **Shared read-only**
  - Initialization Guarantees
  - Immutability
- **Not shared**
  - Stack Confined: Method Local Variables
  - Thread Confined: ThreadLocals

If multiple threads access the same mutable state variable without appropriate synchronization, *your program is broken*. There are three ways to fix it:

- Use synchronization whenever accessing the state variable
- Make the state variable immutable
- Don't share the state variable across threads [JCIP 2]

Immutablility

- **Immutable objects**
  - must be properly constructed
    - this reference does not escape during construction
  - cannot be modified after construction
  - are always thread safe

Strict Immutable

- **All its fields are final**
  - Strongly recommended
- **Can be published in any way**
  - Visibility implies consistency

Effectively Immutable

- **State does not change after creation**
  - But not enforced with final
- **Must be published safely**

Safe Publication

- **How to make objects visible to other threads**
  - Store a reference to it into a volatile variable
  - Store a reference into a field which is guarded by a lock
  - Initialize an object reference in a static initializer
  - Store a reference into a final field of a properly constructed object

  ⇒ **These rules are consequences of the JMM**

- **Example for unsafe publication mechanism**

```
class UnsafeGlobal {
   private static Object reference;
   public static void setValue(Object ref){ reference = ref; }
   public static Object getValue() { return reference; }
}
```

Implementing Immutability

Effective Java: Item 15

1. **Don't provide any methods that modify the object's state**
2. **Ensure that the class can't be extended (final class)**
3. **Make all fields final**
   – Strict immutability => guarantees visibility
4. **Make all fields private**
   – final fields to immutable objects could also be declared public
5. **Ensure exclusive access to any mutable components**
   – Clone references to mutable components passed to the constructor
   – Clone references to mutable components returned by getters

Advantages of Immutable Objects

- **Easy to reason about**
  - Immutable objects are always in exactly one state
  - Method calls on immutables return identical results for identical arguments
- **Easy to implement**
  - After invariant is established on construction, no need to check again
  - No defensive copies, copy constructors and the like
- **Immutable objects are inherently thread safe**
  - There is no change to coordinate, thus they can be shared freely
- **Immutable objects make great building blocks for other objects**
  - Easier to maintain invariant if building blocks don't change
  - Can safely be used as keys for HashMaps

Summary

- **Initialization-Safety-Guarantee and Immutables**
  - Immutable objects **not only** have to
    - declare fields as private
    - copy non-immutable references in constructors
    - copy non-immutable references before returning to the caller
  - Immutable objects **also** have to guarantee
    - that the immutable content is visible to all threads after creation
    - that the instance does not escape during initialization

> Just as it is a good practice to make all fields private unless they need greater visibility [EJ Item 12], **it is a good practice to make all fields final** unless they need to be mutable (which is rarely the case). [JCIP 3.4.1]

Safe Object Sharing

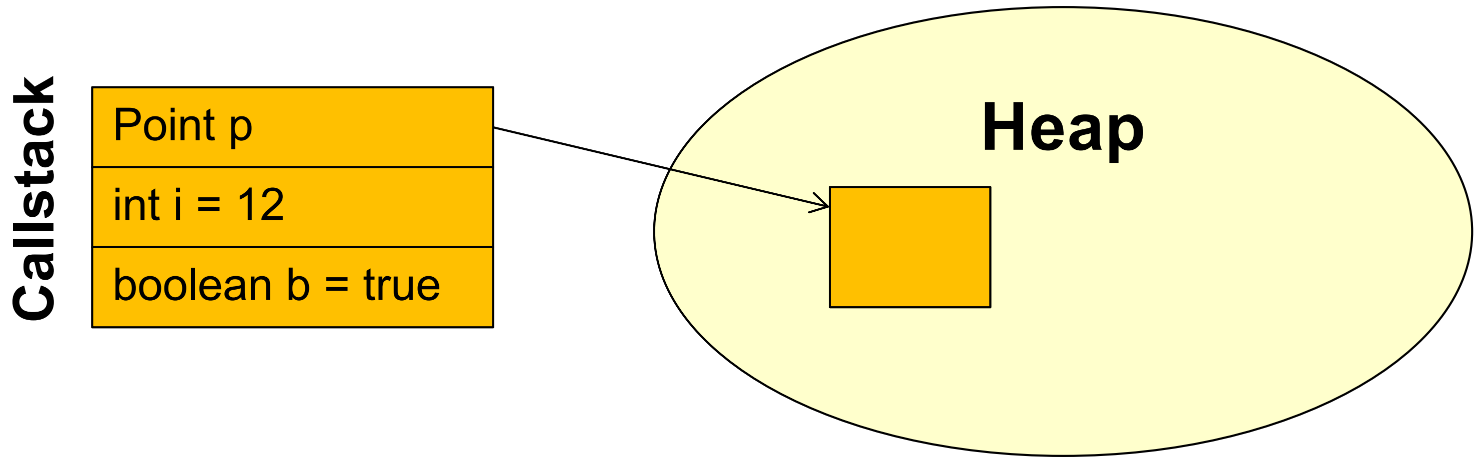
- **Shared read-only**
  - Initialization Guarantees
  - Immutability
- **Not shared**
  - Stack Confined: Method Local Variables
  - Thread Confined: ThreadLocals

If multiple threads access the same mutable state variable without appropriate synchronization, *your program is broken*. There are three ways to fix it:

- Use synchronization whenever accessing the state variable ✓
- Make the state variable immutable ✓
- Don't share the state variable across threads [JCIP 2]

Method Local Variables

- **Variables local to a method can only be accessed by the executing thread**
  - Primitive variables: Impossible to violate stack confinement
  - Object references: Just don't publish them / developers discipline



Example SimpleDateFormat: BAD

- **SimpleDateFormat is not threadsafe**

```
public class BadFormatter {
    private static final SimpleDateFormat sdf
                              = new SimpleDateFormat();

    public static String format(Date d) {
        return sdf.format(d);
    }
}
```

Synchronization
Date formats are **not synchronized**. It is recommended to create separate format instances for each thread. If multiple threads access a format concurrently, it must be synchronized externally. [JavaDoc]

Example SimpleDateFormat: GOOD

- **Solution: Use a fresh instance on every invocation**

```
public class GoodFormatter {
    public static String format(Date d) {
        SimpleDateFormat sdf = new SimpleDateFormat();
        return sdf.format(d);
    }
}
```

  - If this really is a performance problem use a ThreadLocal

Safe Object Sharing

- **Shared read-only**
  - Initialization Guarantees
  - Immutability
- **Not shared**
  - Stack Confined: Method Local Variables
  - Thread Confined: ThreadLocals

If multiple threads access the same mutable state variable without appropriate synchronization, *your program is broken*. There are three ways to fix it:

- Use synchronization whenever accessing the state variable ✓
- Make the state variable immutable ✓
- Don't share the state variable across threads

[JCIP 2]

ThreadLocal

- **Thread-local Variables**
  - A thread-local variable provides a separate copy of its value for each thread that uses it
  - Provides a means to pass state along the call stack without having to explicitly define an additional method parameter
  - ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread (e.g., a user ID or Transaction ID)
- **Class ThreadLocal**

```
class ThreadLocal<T> {
   public T get();                  // returns thread-local value
   public void set(T value);        // sets value for current thread

   protected T initialValue();      // defines an initial value
   public void remove();            // since 1.5
}
```

Example SimpleDateFormat: GOOD

- **One instance per thread**

```
class ThreadLocalFormatter {
    private static ThreadLocal<SimpleDateFormat> local =
        ThreadLocal.withInitial(() -> new SimpleDateFormat());




    public static String format(Date d) {
        return local.get().format(d);
    }
}
```

ThreadLocalRandom

- **java.util.Random**
  - Random is threadsafe
  - The concurrent use of the same Random instance across threads may encounter contention and consequent poor performance
- **java.util.ThreadLocalRandom**
  - A random number generator isolated to the current thread

```
private static final ThreadLocal<ThreadLocalRandom> localRandom =
   new ThreadLocal<ThreadLocalRandom>() {
      protected ThreadLocalRandom initialValue() {
         return new ThreadLocalRandom();
      }
   };
```

ThreadLocal: Simplified Implementation

```
class ThreadLocal<T> {
   private Map<Thread, T> values = new ConcurrentHashMap<Thread, T>();

   public T get() {
      Thread t = Thread.currentThread();
      T value  = values.get(t);
      if(value == null && !values.containsKey(t)) {
         value = initialValue();
         values.put(t, value);
      }
      return value;
   }
   public void set(T value){
      values.put(Thread.currentThread(), value);
   }
   public T initialValue(){ return null; }
}
```

ThreadLocal: Example

```
public class ThreadLocalTest {
   static ThreadLocal<Integer> value = ThreadLocal.withInitial(() -> 0);

   public static void localInc() {
      System.out.println(Thread.currentThread().getName() + ":  "
                                                          + value.get());
      value.set(1 + value.get());
   }

   static class T extends Thread {
      int n;
      T(int n) { this.n = n; }
      public void run() {
         for (int i = 0; i < n; i++) { localInc(); }
      }
   }
```

ThreadLocal: Example

```
    public static void main(String[] args) throws Exception {
        T t1 = new T(3); t1.start();
        T t2 = new T(5); t2.start();
        T t3 = new T(2); t3.start();

        localInc();

        t1.join();
        t2.join();
        t3.join();
    }
}
```

```
Thread-1:  0
Thread-2:  0
Thread-2:  1
main:  0
Thread-0:  0
Thread-0:  1
Thread-0:  2
Thread-1:  1
Thread-1:  2
Thread-1:  3
Thread-1:  4
```

Summary / Advices

- **Do not mutate objects if not required**
  – Instead return copies which reflect the desired changes
- **Do not share objects if not required**
  – Instead keep your objects local to the executing thread

- **People claim performance problems with those approaches**
  – Immutability can even give you a performance boost
    - No synchronization => No flushes, no refreshes
    - Modern GC algorithms => Creation of new short lived objects is super cheap
    - final allows many optimizations by the compiler
  – Write correct code first - then care about performance!

More 'final' => less trouble!