Concurrent Programming: Safety

Christian Kästner    Bogdan Vasilescu
Example: Money-Grab

```java
public class BankAccount {
    private long balance;

    public BankAccount(long balance) {
        this.balance = balance;
    }

    static void transferFrom(BankAccount source, BankAccount dest, long amount) {
        source.balance -= amount;
        dest.balance += amount;
    }

    public long balance() {
        return balance;
    }
}
```
What would you expect this to print?

```java
public static void main(String[] args) throws InterruptedException {
    BankAccount bugs = new BankAccount(100);
    BankAccount daffy = new BankAccount(100);

    Thread bugsThread = new Thread(() -> {
        for (int i = 0; i < 1000000; i++)
            transferFrom(daffy, bugs, 100);
    });

    Thread daffyThread = new Thread(() -> {
        for (int i = 0; i < 1000000; i++)
            transferFrom(bugs, daffy, 100);
    });

    bugsThread.start(); daffyThread.start();
    bugsThread.join(); daffyThread.join();
    System.out.println(bugs.balance() + daffy.balance());
}
```
What went wrong?

• Daffy & Bugs threads were stomping each other
• Transfers did not happen in sequence
• Constituent reads and writes interleaved randomly
• Random results ensued
Fix: Synchronized access (visibility)

@ThreadSafe
public class BankAccount {

    @GuardedBy("this")
    private long balance;

    public BankAccount(long balance) {
        this.balance = balance;
    }

    static synchronized void transferFrom(BankAccount source, BankAccount dest, long amount) {
        source.balance -= amount;
        dest.balance  += amount;
    }

    public synchronized long balance() {
        return balance;
    }
}
Example: serial number generation

What would you expect this to print?

```java
public class SerialNumber {
    private static long nextSerialNumber = 0;

    public static long generateSerialNumber() {
        return nextSerialNumber++;
    }

    public static void main(String[] args) throws InterruptedException {
        Thread threads[] = new Thread[5];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(() -> {
                for (int j = 0; j < 1_000_000; j++)
                    generateSerialNumber();
            });
            threads[i].start();
        }
        for(Thread thread : threads) thread.join();
        System.out.println(generateSerialNumber());
    }
}
```
What went wrong?

• The ++ (increment) operator is not atomic!
  – It reads a field, increments value, and writes it back
• If multiple calls to generateSerialNumber see the same value, they generate duplicates
@ThreadSafe
public class SerialNumber {
    @GuardedBy("this")
    private static int nextSerialNumber = 0;

    public static synchronized int generateSerialNumber() {
        return nextSerialNumber++;
    }

    public static void main(String[] args) throws InterruptedException{
        Thread threads[] = new Thread[5];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(() -> {
                for (int j = 0; j < 1_000_000; j++)
                    generateSerialNumber();
            });
            threads[i].start();
        }
        for(Thread thread : threads) thread.join();
        System.out.println(generateSerialNumber());
    }
}
Part 1: Design at a Class Level
Design for Change: Information Hiding, Contracts, Design Patterns, Unit Testing
Design for Reuse: Inheritance, Delegation, Immutability, LSP, Design Patterns

Part 2: Designing (Sub)systems
Understanding the Problem
Responsibility Assignment, Design Patterns, GUI vs Core, Design Case Studies
Testing Subsystems
Design for Reuse at Scale: Frameworks and APIs

Part 3: Designing Concurrent Systems
Concurrency Primitives, Synchronization
Designing Abstractions for Concurrency
Distributed Systems in a Nutshell

Intro to Java
Git, CI

UML
Static Analysis
Performance

GUIs

More Git

GUIs

Design

Intro to Java

GUIs

Static Analysis
Performance

More Git

GUIs
Learning Goals

• Understand and use Java primitives for concurrency: threads, synchronization, volatile, wait/notify
• Understand problems of undersynchronization and oversynchronization
• Use information hiding to reduce need for synchronization
• Decide on strategy to achieve safety, when and how to synchronize, and use both fine-grained and coarse-grained synchronization as appropriate
JAVA PRIMITIVES:
WAIT, NOTIFY, AND TERMINATION
Guarded Methods

• What to do on a method if the precondition is not fulfilled (e.g., transfer money from bank account with insufficient funds)
  • throw exception (balking)
  • wait until precondition is fulfilled (guarded suspension)
  • wait and timeout (combination of balking and guarded suspension)
Example: Balking

- If there are multiple calls to the job method, only one will proceed while the other calls will return with nothing.

```java
public class BalkingExample {
    private boolean jobInProgress = false;

    public void job() {
        synchronized (this) {
            if (jobInProgress) { return; }
            jobInProgress = true;
        }
        // Code to execute job goes here
    }

    void jobCompleted() {
        synchronized (this) {
            jobInProgress = false;
        }
    }
}
```
Guarded Suspension

• Block execution until a given condition is true
• For example,
  – pull element from queue, but wait on an empty queue
  – transfer money from bank account as soon sufficient funds are there
• Blocking as (often simpler) alternative to callback
Monitor Mechanics in Java

- **Object.wait()** – suspends the current thread’s execution, releasing locks
- **Object.wait(timeout)** – suspends the current thread’s execution for up to *timeout* milliseconds
- **Object.notify()** – resumes one of the waiting threads
- See documentation for exact semantics
Example: Guarded Suspension

- Loop until condition is satisfied
  - wasteful, since it executes continuously while waiting

```java
public void guardedJoy() {
    // Simple loop guard. Wastes
    // processor time. Don't do this!
    while (!joy) {
    }
    System.out.println("Joy has been achieved!");
}
```
Example: Guarded Suspension

• More efficient: invoke Object.wait to suspend current thread

```java
public synchronized guardedJoy() {
    while(!joy) {
        try {
            wait();
        } catch (InterruptedException e) {} 
    }
    System.out.println("Joy and efficiency have been achieved!");
}
```

• When wait is invoked, the thread releases the lock and suspends execution. The invocation of wait does not return until another thread has issued a notification

```java
public synchronized notifyJoy() {
    joy = true;
    notifyAll();
}
```
Never invoke wait outside a loop!

• Loop tests condition before and after waiting

• Test before skips wait if condition already holds
  – Necessary to ensure liveness
  – Without it, thread can wait forever!

• Testing after wait ensures safety
  – Condition may not be true when thread wakes
  – If thread proceeds with action, it can destroy invariants!
All of your waits should look like this

synchronized (obj) {
    while (<condition does not hold>) {
        obj.wait();
    }

    ... // Perform action appropriate to condition
}
Why can a thread wake from a wait when condition does not hold?

- Another thread can slip in between notify & wake

- Another thread can invoke notify accidentally or maliciously when condition does not hold
  - This is a flaw in java locking design!
  - Can work around flaw by using private lock object

- Notifier can be liberal in waking threads
  - Using notifyAll is good practice, but causes this

- Waiting thread can wake up without a notify(!)
  - Known as a *spurious wakeup*
Guarded Suspension vs Balking

• Guarded suspension:
  – Typically only when you know that a method call will be suspended for a finite and reasonable period of time
  – If suspended for too long, the overall program will slow down

• Balking:
  – Typically only when you know that the method call suspension will be indefinite or for an unacceptably long period
Monitor Example

class SimpleBoundedCounter {
    protected long count = MIN;
    public synchronized long count() { return count; }

    public synchronized void inc() throws InterruptedException {
        awaitUnderMax(); setCount(count + 1); }

    public synchronized void dec() throws InterruptedException {
        awaitOverMin(); setCount(count - 1); }

    protected void setCount(long newValue) { // PRE: lock held
        count = newValue;
        notifyAll(); // wake up any thread depending on new value
    }

    protected void awaitUnderMax() throws InterruptedException {
        while (count == MAX) wait();
    }

    protected void awaitOverMin() throws InterruptedException {
        while (count == MIN) wait();
    }
}
Interruption

• Difficult to kill threads once started, but may politely ask to stop (thread.interrupt())

• Long-running threads should regularly check whether they have been interrupted

• Threads waiting with wait() throw exceptions if interrupted

• Read documentation

```java
public class Thread {
    public void interrupt() { ... }
    public boolean isInterrupted() { ... }
    ...
}
```
class PrimeProducer extends Thread {
    private final BlockingQueue<BigInteger> queue;
    PrimeProducer(BlockingQueue<BigInteger> queue) {
        this.queue = queue;
    }
    public void run() {
        try {
            BigInteger p = BigInteger.ONE;
            while (!Thread.currentThread().isInterrupted())
                queue.put(p = p.nextProbablePrime());
        } catch (InterruptedException consumed) {
            /* Allow thread to exit */
        }
    }
    public void cancel() { interrupt(); }
}

For details, see Java Concurrency In Practice, Chapter 7
BUILDING HIGHER LEVEL CONCURRENCY MECHANISMS
Beyond Java Primitives

• Java Primitives (synchronized, wait, notify) are low level mechanisms
• For most tasks better higher-level abstractions exist
• Writing own abstractions is possible, but potentially dangerous – use libraries written by experts
Example: read-write locks (API)
*Also known as shared/exclusive mode locks*

- If multiple threads are accessing an object for reading data, no need to use a synchronized block (or other mutually exclusive locks)

```java
private final RwLock lock = new RwLock();

lock.readLock();
try {
    // Do stuff that requires read (shared) lock
} finally {
    lock.unlock();
}

lock.writeLock();
try {
    // Do stuff that requires write (exclusive) lock
} finally {
    lock.unlock();
}
```
Example: read-write locks (Impl. 1/2)

```java
public class RwLock {
    // State fields are protected by RwLock's intrinsic lock

    /** Num threads holding lock for read. */
    @GuardedBy("this")
    private int numReaders = 0;

    /** Whether lock is held for write. */
    @GuardedBy("this")
    private boolean writeLocked = false;

    public synchronized void readLock() throws InterruptedException {
        while (writeLocked) {
            wait();
        }
        numReaders++;
    }
}
```
Example: read-write locks (Impl. 2/2)

```java
public synchronized void writeLock() throws InterruptedException {
    while (numReaders != 0 || writeLocked) {
        wait();
    }
    writeLocked = true;
}

public synchronized void unlock() {
    if (numReaders > 0) {
        numReaders--;
    } else if (writeLocked) {
        writeLocked = false;
    } else {
        throw new IllegalStateException("Lock not held");
    }
    notifyAll(); // Wake any waiters
}
```
Caveat: RwLock is just a toy!

• It has poor fairness properties
  – Readers can starve writers!

• java.util.concurrent provides an industrial strength ReadWriteLock

• More generally, avoid wait/notify
  – In the early days it was all you had
  – Nowadays, higher level concurrency utils are better
Summary

• Concurrency for exploiting multiple processors, simplifying modeling, simplifying asynchronous events
• Safety, liveness and performance hazards matter
• Synchronization on any Java object; volatile ensures visibility
• Wait/notify for guards, interruption for cancelation – building blocks for higher level abstractions
THREAD SAFETY: DESIGN TRADEOFFS
Recall: Synchronization for Safety

• If multiple threads access the same mutable state variable without appropriate synchronization, the program is broken.

• There are three ways to fix it:
  – Don't share the state variable across threads;
  – Make the state variable immutable; or
  – Use synchronization whenever accessing the state variable.
Thread Confinement

• Ensure variables are not shared across threads (concurrency version of encapsulation)
• Stack confinement:
  – Object only reachable through local variables (never leaves method) \(\rightarrow\) accessible only by one thread
  – Primitive local variables always thread-local
• Confinement across methods/in classes needs to be done carefully (see immutability)
Example: Thread Confinement

- Shared ark object
- TreeSet is not thread safe but it’s local → can’t leak
- Defensive copying on AnimalPair

```java
public int loadTheArk(Collection<Animal> candidates) {
    SortedSet<Animal> animals;
    int numPairs = 0;
    Animal candidate = null;
    // animals confined to method, don't let them escape!
    animals = new TreeSet<Animal>(new SpeciesGenderComparator());
    animals.addAll(candidates);
    for (Animal a : animals) {
        if (candidate == null || !candidate.isPotentialMate(a))
            candidate = a;
        else {
            ark.load(new AnimalPair(candidate, a));
            ++numPairs;
            candidate = null;
        }
    }
    return numPairs;
}
```
Confinement with ThreadLocal

• ThreadLocal holds a separate value for each cache (essentially Map<Thread,T>)
  – create variables that can only be read and written by the same thread
  – if two threads are executing the same code, and the code has a reference to a ThreadLocal variable, then the two threads cannot see each other's ThreadLocal variables
Example: ThreadLocal

```java
public static class MyRunnable implements Runnable {
    private ThreadLocal<Integer> threadLocal = new ThreadLocal<Integer>();

    @Override
    public void run() {
        threadLocal.set((int) (Math.random() * 100D));
        System.out.println(threadLocal.get());
    }
}

public static void main(String[] args) throws InterruptedException {
    MyRunnable sharedRunnableInstance = new MyRunnable();

    Thread thread1 = new Thread(sharedRunnableInstance);
    Thread thread2 = new Thread(sharedRunnableInstance);

    thread1.start();
    thread2.start();

    thread1.join(); // wait for thread 1 to terminate
    thread2.join(); // wait for thread 2 to terminate
}
```

From: http://tutorials.jenkov.com/java-concurrency/threadlocal.html
Immutable Objects

• Immutable objects can be shared freely
• Remember:
  – Fields initialized in constructor
  – Fields final
  – Defensive copying if mutable objects used internally
Synchronization

• **Thread-safe** objects vs **guarded**:  
  – Thread-safe objects perform synchronization internally (clients can always call safely)  
  – Guarded objects require clients to acquire lock for safe calls

• Thread-safe objects are idiot-proof to use, but guarded objects can be more flexible
Designing Thread-Safe Objects

• Identify variables that represent the object’s state
  – may be distributed across multiple objects
• Identify invariants that constraint the state variables
  – important to understand invariants to ensure atomicity of operations
• Establish a policy for managing concurrent access to state
What would you change here?

```java
@ThreadSafe
public class PersonSet {
    @GuardedBy("this")
    private final Set<Person> mySet = new HashSet<Person>();

    @GuardedBy("this")
    private Person last = null;

    public synchronized void addPerson(Person p) {
        mySet.add(p);
    }

    public synchronized boolean containsPerson(Person p) {
        return mySet.contains(p);
    }

    public synchronized void setLast(Person p) {
        this.last = p;
    }
}
```
Coarse-Grained Thread-Safety

- Synchronize all access to all state with the object

```java
@ThreadSafe
public class PersonSet {
    @GuardedBy("this")
    private final Set<Person> mySet = new HashSet<Person>();

    @GuardedBy("this")
    private Person last = null;

    public synchronized void addPerson(Person p) {
        mySet.add(p);
    }

    public synchronized boolean containsPerson(Person p) {
        return mySet.contains(p);
    }

    public synchronized void setLast(Person p) {
        this.last = p;
    }
}
```
Fine-Grained Thread-Safety

• “Lock splitting”: Separate state into independent regions with different locks

```java
@ThreadSafe
public class PersonSet {
    @GuardedBy("myset")
    private final Set<Person> mySet = new HashSet<Person>();

    @GuardedBy("this")
    private Person last = null;

    public void addPerson(Person p) {
        synchronized (mySet) {
            mySet.add(p);
        }
    }

    public boolean containsPerson(Person p) {
        synchronized (mySet) {
            return mySet.contains(p);
        }
    }

    public synchronized void setLast(Person p) {
        this.last = p;
    }
}
```
Private Locks

• Any object can serve as lock

```java
@ThreadSafe
public class PersonSet {
    @GuardedBy("myset")
    private final Set<Person> mySet = new HashSet<Person>();

    private final Object myLock = new Object();
    @GuardedBy("myLock")
    private Person last = null;

    public void addPerson(Person p) {
        synchronized (mySet) {
            mySet.add(p);
        }
    }

    public synchronized boolean containsPerson(Person p) {
        synchronized (mySet) {
            return mySet.contains(p);
        }
    }

    public void setLast(Person p) {
        synchronized (myLock) {
            this.last = p;
        }
    }
}
```
Delegating thread-safety to well designed classes

• Recall previous CountingFactorizer

```java
@NotThreadSafe
public class CountingFactorizer implements Servlet {
    private long count = 0;

    public long getCount() { return count; }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        ++count;
        encodeIntoResponse(resp, factors);
    }
}
```
Delegating thread-safety to well designed classes

• Replace long counter with an AtomicLong

```java
@ThreadSafe
public class CountingFactorizer implements Servlet {
    private final AtomicLong count = new AtomicLong(0);

    public long getCount() { return count.get(); }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        count.incrementAndGet();
        encodeIntoResponse(resp, factors);
    }
}
```
Synchronize only relevant method parts

• Design heuristic:
  – Get in, get done, and get out
    • Obtain lock
    • Examine shared data
    • Transform as necessary
    • Drop lock
  – If you must do something slow, move it outside synchronized region
Example: What to synchronize?

```
@ThreadSafe
public class AttributeStore {
    @GuardedBy("this")
    private final Map<String, String> attributes = new HashMap<String, String>();

    public synchronized boolean userLocationMatches(String name, String regexp) {
        String key = "users." + name + ".location";
        String location = attributes.get(key);
        if (location == null)
            return false;
        else
            return Pattern.matches(regexp, location);
    }
}
```
Narrowing lock scope

```java
@ThreadSafe
public class BetterAttributeStore {
    @GuardedBy("this")
    private final Map<String, String> attributes = new HashMap<String, String>();

    public boolean userLocationMatches(String name, String regexp) {
        String key = "users." + name + ".location";
        String location;
        synchronized (this) {
            location = attributes.get(key);
        }
        if (location == null)
            return false;
        else
            return Pattern.matches(regexp, location);
    }
}
```
Fine-Grained vs Coarse-Grained Tradeoffs

• Coarse-Grained is simpler

• Fine-Grained allows concurrent access to different parts of the state

• When invariants span multiple variants, fine-grained locking needs to ensure that all relevant parts are using the same lock or are locked together

• Acquiring multiple locks requires care to avoid deadlocks
Over vs Undersynchronization

• Undersynchronization -> safety hazard
• Oversynchronization -> liveness hazard and reduced performance
Guards and Client-Side Locking

Where is the issue?

```java
public class ListHelper<E> {
    public List<E> list =
        Collections.synchronizedList(new ArrayList<E>());
    ...
    public synchronized boolean putIfAbsent(E x) {
        boolean absent = !list.contains(x);
        if (absent)
            list.add(x);
        return absent;
    }
}
```
Guards and Client-Side Locking

• Synchronize on target:

```java
public class ListHelper<E> {
    public List<E> list =
        Collections.synchronizedList(new ArrayList<E>());
    ...
    public boolean putIfAbsent(E x) {
        synchronize(list) {
            boolean absent = !list.contains(x);
            if (absent)
                list.add(x);
            return absent;
        }
    }
}
```
Avoiding deadlock

• Deadlock caused by a cycle in waits-for graph
  – T1: synchronized(a){ synchronized(b){ ... } }
  – T2: synchronized(b){ synchronized(a){ ... } }

• To avoid these deadlocks:
  – When threads have to hold multiple locks at the same time, **all threads obtain locks in same order**
Summary of policies:

• **Thread-confined.** A thread-confined object is owned exclusively by and confined to one thread, and can be modified by its owning thread.

• **Shared read-only.** A shared read-only object can be accessed concurrently by multiple threads without additional synchronization, but cannot be modified by any thread. Shared read-only objects include immutable and effectively immutable objects.

• **Shared thread-safe.** A thread-safe object performs synchronization internally, so multiple threads can freely access it through its public interface without further synchronization.

• **Guarded.** A guarded object can be accessed only with a specific lock held. Guarded objects include those that are encapsulated within other thread-safe objects and published objects that are known to be guarded by a specific lock.
Tradeoffs

• Strategies:
  – Don't share the state variable across threads;
  – Make the state variable immutable; or
  – Use synchronization whenever accessing the state variable.
    • Thread-safe vs guarded
    • Coarse-grained vs fine-grained synchronization

• When to choose which strategy?
  – Avoid synchronization if possible
  – Choose simplicity over performance where possible
Documentation

• Document a class's thread safety guarantees for its clients
• Document its synchronization policy for its maintainers.
• @ThreadSafe, @GuardedBy annotations not standard but useful
REUSE RATHER THAN BUILD: KNOW THE LIBRARIES
java.util.concurrent is BIG (1)

• Atomic vars - java.util.concurrent.atomic
  – Support various atomic read-modify-write ops
• Executor framework
  – Tasks, futures, thread pools, completion service, etc.
• Locks - java.util.concurrent.locks
  – Read-write locks, conditions, etc.
• Synchronizers
  – Semaphores, cyclic barriers, countdown latches, etc.
java.util.concurrent is BIG (2)

• Concurrent collections
  – Shared maps, sets, lists
• Data Exchange Collections
  – Blocking queues, deques, etc.
• Pre-packaged functionality - java.util.arrays
  – Parallel sort, parallel prefix
Parallel Collections

• Java 1.2: Collections.synchronizedMap(map)
• Java 5: ConcurrentHashMap
  – putIfAbsent, replace, ... built in
  – Fine-grained synchronization
• BlockingQueue, CopyOnWriteArrayList, ...
Summary

- Three design strategies for achieving safety: Thread locality, immutability and synchronization
- Tradeoffs for synchronization
  - thread-safe vs guarding
  - fine-grained vs coarse-grained
  - simplicity vs performance
- Avoiding deadlocks
- Reuse rather than build abstractions; know the libraries
Recommended Readings

• Goetz et al. Java Concurrency In Practice. Pearson Education, 2006, Chapters 2-5, 11