Principles of Software Construction: Objects, Design, and Concurrency

Concurrency: Safety & Immutability

Claire Le Goues **Vincent Hellendoorn**

Carnegie Mellon University School of Computer Science

institute for SOFTWARE RESEARCH

Revisiting callbacks

Recall: What if my Thread isn't Alone?

- In JS event loops:
  - Waiting is synchronous
  - Each message is processed fully without interruption
- What if we wanted multiple threads?
  - For parallelism
  - Multiple users on a website

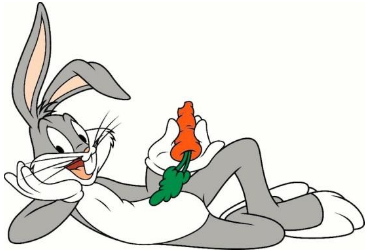
Remember the money-grab example?

```java
public static void main(String[] args) throws InterruptedException {
    BankAccount bugs = new BankAccount(1_000_000);
    BankAccount daffy = new BankAccount(1_000_000);

    Thread bugsThread = new Thread(()-> {
        for (int i = 0; i < 1_000_000; i++)
            transferFrom(daffy, bugs, 1);
    });

    Thread daffyThread = new Thread(()-> {
        for (int i = 0; i < 1_000_000; i++)
            transferFrom(bugs, daffy, 1);
    });

    bugsThread.start(); daffyThread.start();
    bugsThread.join(); daffyThread.join();
    System.out.println(bugs.balance() - daffy.balance());
}
```



Last Week

- Concurrency hazards:
  - Safety
  - Liveness
  - Performance

Safety, Liveness, Performance

CONCURRENCY HAZARDS

Quick Recap

1. Safety Hazard

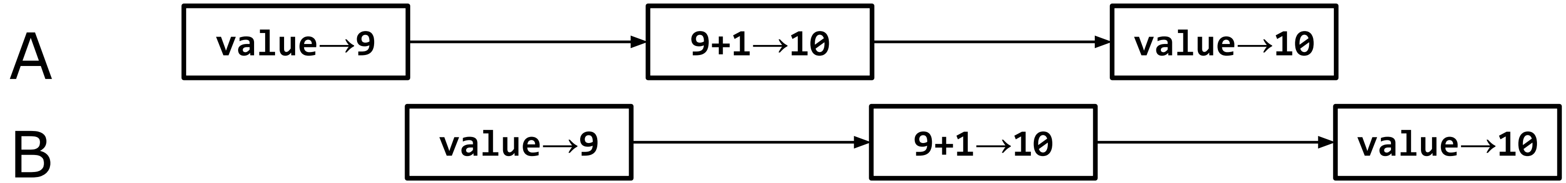
- The ordering of operations in multiple threads is **unpredictable**.

```java
@NotThreadSafe
public class UnsafeSequence {
    private int value;

    public int getNext() {
        return value++;
    }
}
```

Not atomic

- Unlucky execution of `UnsafeSequence.getNext`



2. Liveness Hazard

- Safety: “nothing bad ever happens”
- Liveness: “something good eventually happens”
- Deadlock
  - Infinite loop in sequential programs
  - Thread A waits for a resource that thread B holds exclusively, and B never releases it → A will wait forever
    - E.g., Dining philosophers
- Elusive: depend on relative timing of events in different threads

3. Performance Hazard

- Liveness: “something good eventually happens”
- Performance: we want something good to happen quickly

- Multi-threading involves runtime overhead:
  - Coordinating between threads (locking, signaling, memory sync)
  - Context switches
  - Thread creation & teardown
  - Scheduling
- Not all problems can be solved faster with more resources
  - One mother delivers a baby in 9 months

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.

Outlook

- Concurrency hazards:
  - Safety
  - Liveness
  - Performance
- Today:
  - Immutability
  - Thread Confinement
  - Java primitives
    - For ensuring visibility, atomicity
    - Waiting
    - With some discussion of other languages

How to Prevent Competing Access?

- Anyone remember the simple solutions?

How to Prevent Competing Access?

- Anyone remember the simple solutions?
  - Don’t have state!
  - Don’t have shared state!
  - Don’t have shared **mutable** state!

Today

- **Immutability**
- Thread Confinement
- Java primitives
  - For ensuring visibility, atomicity
  - Waiting
  - With some discussion of other languages

Immutability

- A key concept in design, not just for concurrency
  - Inherently Thread-safe
  - No risks in sharing
  - Can make things very simple

Making a Class Immutable

```
public class Complex {
    final double re, im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    public double getRealPart()      { return re; }
    public double getImaginaryPart() { return im; }

    public double setRealPart(double re)      { this.re = re; }
    public double setImaginaryPart(double im) { this.im = im; }

    …
```

Making a Class Immutable

```
public final class Complex {
    private final double re, im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    // Getters without corresponding setters
    public double getRealPart()      { return re; }
    public double getImaginaryPart() { return im; }

    ...
```

Ensuring Immutability

- Don’t provide any mutators
- Ensure that no methods may be overridden
- Make all fields final
- Make all fields private
- Ensure security of any mutable components

Immutability

What if you need to make a change?

```
public final class Complex {
    private final double re, im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    // Getters without corresponding setters
    public double getRealPart()      { return re; }
    public double getImaginaryPart() { return im; }

    public ??? add(Complex c) {
       ...
    }
```

Making a Class Immutable

```
public final class Complex {
    private final double re, im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    // Getters without corresponding setters
    public double getRealPart()      { return re; }
    public double getImaginaryPart() { return im; }

    // subtract, multiply, divide similar to add
    public Complex add(Complex c) {
        return new Complex(re + c.re, im + c.im);
    }
```

Immutability

We have seen this before! Is Game truly immutable?

```java
    public Game play(int x, int y) {
        if (this.board.getCell(x, y) != null)
            return this;
        if (this.getWinner() != null)
            return this;
        List<Game> newHistory = new ArrayList<>(this.history);
        newHistory.add(this);
        Player nextPlayer = this.player == Player.PLAYER0 ? Player.PLAYER1 : Player.PLAYER0;
        return new Game(this.board.updateCell(x, y, this.player), nextPlayer, newHistory);
    }
```

Immutability

What functionality was made really easy by this design?

```
    public Game play(int x, int y) {
        if (this.board.getCell(x, y) != null)
            return this;
        if (this.getWinner() != null)
            return this;
        List<Game> newHistory = new ArrayList<>(this.history);
        newHistory.add(this);
        Player nextPlayer = this.player == Player.PLAYER0 ? Player.PLAYER1 : Player.PLAYER0;
        return new Game(this.board.updateCell(x, y, this.player), nextPlayer, newHistory);
    }
```

Immutable?

```
class Stack {
    readonly #inner: any[]
    constructor (inner: any[]) {
        this.#inner=inner
    }
    push(o: any): Stack {
        const newInner = this.#inner.slice()
        newInner.push(o)
        return new Stack(newInner)
    }
    peek(): any {
        return this.#inner[this.#inner.length-1]
    }
    getInner(): any[] {
        return this.#inner
    }
}
```

Immutable?

Inner mutable state (List in Java)

Create copy of mutable object (new ArrayList(old) in Java)

Return new immutable object

```
class Stack {
    readonly #inner: any[]
    constructor (inner: any[]) {
        this.#inner=inner
    }
    push(o: any): Stack {
        const newInner = this.#inner.slice()
        newInner.push(o)
        return new Stack(newInner)
    }
    peek(): any {
        return this.#inner[this.#inner.length-1]
    }
    getInner(): any[] {
        return this.#inner
    }
}
```

Aliasing is what makes Mutable State risky

Many variables may point to same object

Any reference to the object can modify the object, effect seen by all other users

x, y, and z all point to the same mutable array

```
const x = [ 1, 2, 3 ]
const y = x
function foo(z: number[]): void { /*...*/ }
foo(y)
```

Immutable?

Inner mutable state (List in Java)

Create copy of mutable object (new ArrayList(old) in Java)

Return new immutable object

Leak mutable state

Accept mutable state

```
class Stack {
    readonly #inner: any[]
    constructor (inner: any[]) {
        this.#inner=inner
    }
    push(o: any): Stack {
        const newInner = this.#inner.slice()
        newInner.push(o)
        return new Stack(newInner)
    }
    peek(): any {
        return this.#inner[this.#inner.length-1]
    }
    getInner(): any[] {
        return this.#inner
    }
}
```

Fixed

```
class Stack {
    readonly #inner: any[]
    constructor (inner: any[]) {
        this.#inner=inner.slice()
    }
    push(o: any): Stack {
        const newInner = this.#inner.slice()
        newInner.push(o)
        return new Stack(newInner)
    }
    peek(): any {
        return this.#inner[this.#inner.length-1]
    }
    getInner(): any[] {
        return this.#inner.slice()
        // Java: return new ArrayList(inner)
    }
}
```

Ensuring Immutability

- Don’t provide any mutators
- Ensure that no methods may be overridden
- Make all fields final
- Make all fields private
- **Ensure security of any mutable components**

Writing Immutable Data Structures

Any “set” operation returns a new copy of an object
(can point to old object to save memory, e.g. linked lists)

Final fields of immutable objects are save (e.g., strings, numbers)

Fields of mutable objects must be protected
(encapsulation, making copies)

Careful with mutable constructor/method arguments (make copies)

Easy to make mistakes when mixing mutable and immutable data structures, only academic tools for checking

Trend toward immutable data structures

Immutable data structures common in functional programming

Many recent languages and libraries embrace immutability

- Scala, Rust, stream, React, Java Records

Simplifies building concurrent and distributed systems

Requires some practice when used to imperative programming with mutable state, but will become natural

Circular references & Caching

Immutable data structures often from a directed acyclic graph

Cycles challenging

Cycles often useful for performance (caching)

```
class TreeNode {
   readonly #parent: TreeNode
   readonly #children: TreeNode[]
   constructor(parent: TreeNode,
               children: TreeNode[]) {
       this.#parent = parent
       this.#children = children
   }
   addChild(child: TreeNode) {
       const newChildren = this.#children.slice()
       //const newChild = child.setParent(this)  ??
       newChildren.push(child)
       const newNode = new TreeNode(this.#parent,
                                    newChildren)
       //child.setParent(newNode)  ??
       return newNode
   }
}
```

Design Discussion

Design for Understandability / Maintainability

- Immutable objects are easy to reason about, they won’t change
- Mutable objects have more complicated contracts, function and client both can modify state
- Do not need to think about corner cases of concurrent modification

Design for Reuse

- Easy to reuse even in concurrent settings

Java 16 Records

Records are (shallowly) immutable

No setters

But also no defensive copying of mutable fields

Immutability

Any disadvantages?

Immutability

Any disadvantages?

```java
String x = "It was the best of times, .."; // An entire book.
x += "The end.";
```

Immutability

Any disadvantages?

```
String x = "It was the best of times, .."; // An entire book.
x += "The end.";
```

- For performance reasons, when needed:
  - Provide mutable helpers (e.g. StringBuilder).
  - Bundle common actions

Designing for Immutability

In short: make things immutable unless you really can't

- Especially, smaller data-classes
- Not realistic for classes whose state naturally changes
  - BankAccount: return a new account for each transaction?
  - In that case, minimize mutable part, guard against sharing

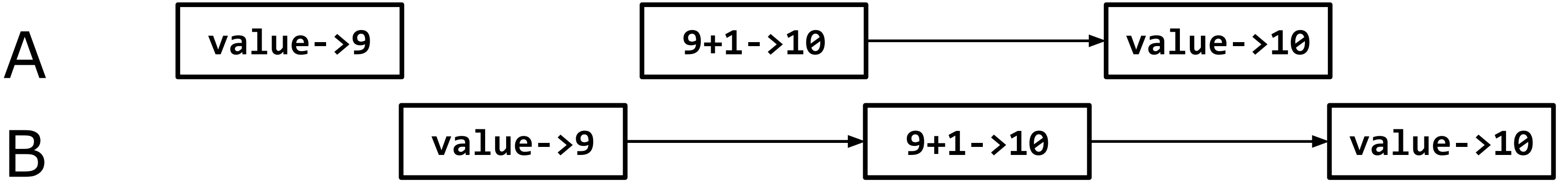
How to Prevent Competing Access?

- Anyone remember the simple solutions?
  - Don’t have state!
  - Don’t have **shared** state!
  - Don’t have shared mutable state!

Continuing

JAVA PRIMITIVES: ENSURING VISIBILITY AND ATOMICITY

Non atomicity and thread (un)safety




```
@NotThreadSafe
public class UnsafeCountingFactorizer 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);
    }
}
```

Cooperative thread termination

How long would you expect this to run?

```
public class StopThread {
    private static boolean stopRequested;

    public static void main(String[] args) throws Exception {
        Thread backgroundThread = new Thread(() -> {
            while (!stopRequested)
                /* Do something */ ;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}
```

What could have gone wrong?

- In the absence of synchronization, there is no guarantee as to when, if ever, one thread will see changes made by another!
  - VMs can and do perform this optimization (“hoisting”):

```
while (!done)
        /* do something */ ;
```

becomes:

```
if (!done)
    while (true)
        /* do something */ ;
```

How do you fix it?

```
public class StopThread {
    private static boolean stopRequested;

    private static synchronized void requestStop() {
        stopRequested = true;
    }

    private static synchronized boolean stopRequested() {
        return stopRequested;
    }

    public static void main(String[] args) throws Exception {
        Thread backgroundThread = new Thread(() -> {
            while (!stopRequested())
                /* Do something */ ;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        requestStop();
    }
}
```

You can do better (?)

`volatile` *is synchronization without mutual exclusion*

```
public class StopThread {
    private static volatile boolean stopRequested;

    public static void main(String[] args) throws Exception {
        Thread backgroundThread = new Thread(() -> {
            while (!stopRequested)
                /* Do something */ ;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}
```

forces all accesses (read or write) to the volatile variable to occur in main memory, effectively keeping the volatile variable out of CPU caches.

https://stackoverflow.com/questions/3519664/difference-between-volatile-and-synchronized-in-java

Volatile Keyword

- Tells compiler and runtime that variable is shared and operations on it should not be reordered with other memory ops
  - A read of a volatile variable always returns the most recent write by any thread
- Volatile is not a substitute for synchronization
  - Volatile variables can only guarantee visibility
  - Locking can guarantee both visibility and atomicity

Summary: Synchronization

- Ideally, avoid shared mutable state
- If you can’t avoid it, synchronize properly
  - Failure to do so causes safety and liveness failures
  - If you don’t sync properly, your program won’t work
- Even atomic operations require synchronization
  - e.g., stopRequested = true
  - And some things that look atomic aren’t (e.g., val++)

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?

Guarded Methods

- What to do on a method if the precondition is not fulfilled
    - E.g., transfer money from bank account with insufficient funds?
    - Obvious in a synchronous world: throw exception (balking)

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 Methods

- What to do on a method if the precondition is not fulfilled
  - Not so obvious in a concurrent world. E.g., suppose a money transfer involves an intermediate/temporary account. One actors sends money to that account & the other pulls it from there. What should the second do when they send a query and the account is empty?

Guarded Methods

- What to do on a method if the precondition is not fulfilled
  - throw exception (balking)
  - wait until precondition is fulfilled (guarded suspension)
  - wait and timeout (combination of balking and guarded suspension)

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 (sometimes simpler) alternative to callback

Example: Guarded Suspension

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

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

Monitor Mechanisms 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

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

```
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

```
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 wakens
    - If thread proceeds with action, it can destroy invariants!

All 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 Design Decisions

- 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

```
public class Thread {
    public void interrupt() { ... }
    public boolean isInterrupted() { ... }
    ...
}
```

Interruption Example

```
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

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) → 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

```
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;
}
```

Approaches to Thread Confinement

- Local variables + defense copying works in most environments
  - Java also has ThreadLocal, to make values accessible to individual threads only
- Other languages require different treatments:
  - JS obviously does not have this problem
  - Python has explicit separation between multi-threading and multi-processing. The latter cannot share state except through special objects

Immutability Simplifies Thread Confinement

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

Synchronization Is More Powerful Still

- But requires explicit locking
- 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 easier to use (harder to misuse), but guarded objects can be more flexible

When Possible, Use The Core Library!

There are well-designed, often fast objects for almost any application in most languages

```
@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);
    }
```

When Possible, Use The Core Library!

There are well-designed, often fast objects for almost any application in most languages – e.g., AtomicLong

```
@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);
    }
}
```

Does Threading Only Complicate Things?

- Not at all!
  - Obviously useful for parallelism and asynchronous I/O
  - But we can also use it for **good design**.
- Threads map to tasks
  - Commonly assign one thread per task
  - Convenient abstract for handling large workloads
- Help manage complex event loops
  - Message passed from one handle to another in single-threaded envs.

Forming Design Patterns

- We've seen:

  Concurrency strategies:
  - Function-based dispatch (callbacks)
  - Using queues to manage asynchronous events

  Thread-safety strategies:
  - Immutability where possible
  - Synchronization on mutable state

Tradeoffs & Summary

- Strategies:
  - Don't share a 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