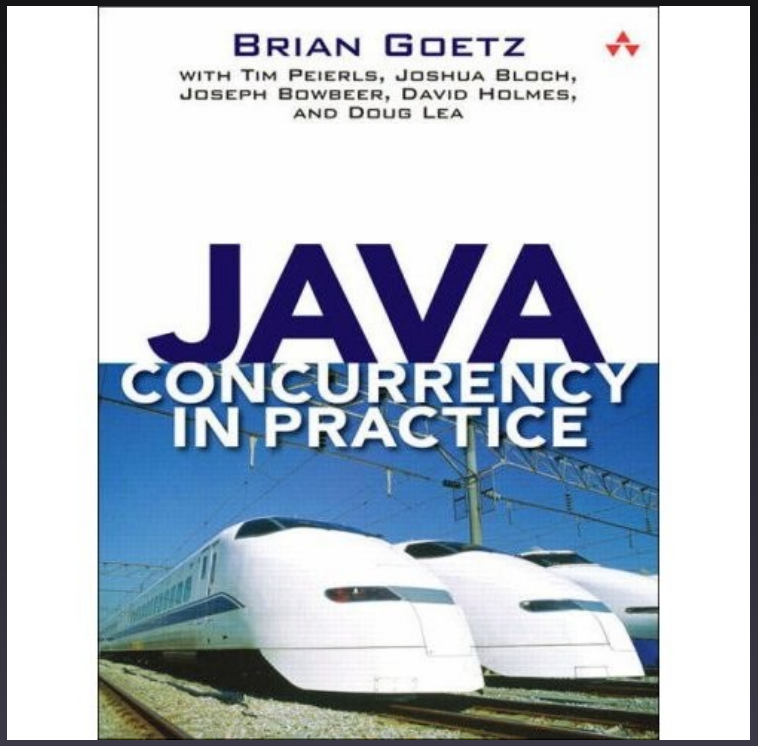
# Java Concurrency in practice




## Chapters: 1,2, 3 & 4

Bjørn Christian Sebak (bse069@student.uib.no)

Karianne Berg (karianne@ii.uib.no)

## INF329 – Spring 2007

# Chapter 1 - Introduction

# Brief history of concurrency

- Before OS, a computer executed a single program from start to finnish
- But running a single program at a time is an inefficient use of computer hardware
- Therefore all modern OS run multiple programs (in seperate *processes*)

# Brief history of concurrency (2)

- Factors for running multiple processes:
  - Resource utilization: While one program waits for I/O, why not let another program run and avoid wasting CPU cycles?
  - Fairness: Multiple users/programs might have equal claim of the computers resources. Avoid having single large programs „hog“ the machine.
  - Convenience: Often desirable to create smaller programs that perform a single task (and coordinate them), than to have one large program that do ALL the tasks

# What is a thread?

- A „lightweight process“ - each process can have many threads
- Threads allow multiple streams of program flow to coexits in a single process.
- While a thread share process-wide resources like memory and files with other threads, they all have their own program counter, stack and local variables

# Benefits of threads

1) Exploiting multiple processors
2) Simplicity of modeling
3) Simplified handling of asynchronous events
4) More responsive user interfaces

# Benefits of threads (2)

- Exploiting multiple processors
  - The processor industry is currently focusing on increasing number of cores on a single CPU rather than increasing clock speed.
  - Well-designed programs with multiple threads can execute simultaneously on multiple processors, increasing resource utilization.
  - Threads are beneficial even for single-processor systems (another thread can run while the first waits for I/O)

# Benefits of threads (3)

- Simplicity of modelling
  - Threads can help decompose large, complex programs into smaller units, while still offering the illusion of having a single, sequential program.
  - Makes it easier to implement and maintain the application, and might give a performance bonus.

# Benefits of threads (4)

- Simplified handling of asynchronous events
  - Imagine a server application in a single-threaded application. If it tries to read from a socket when no data is available – read blocks
  - This stalls not only the current request, but *all other requests* in the system.
  - When each request is run in its own thread, other requests are still processed while one request waits for I/O

# Benefits of using threads (5)

- More responsive user interfaces
  - Single-threaded GUI apps tends to „freeze“ when executing time-consuming operations
  - By excecuting the operation in its own thread, the GUI can still be used while this operation runs
  - Using threads can improve responsiveness and efficienty of GUI applications in java frameworks like Swing/AWT

# Risk of threads

- Important issues to be aware of:

  1)Safety hazards

  2)Liveness hazards

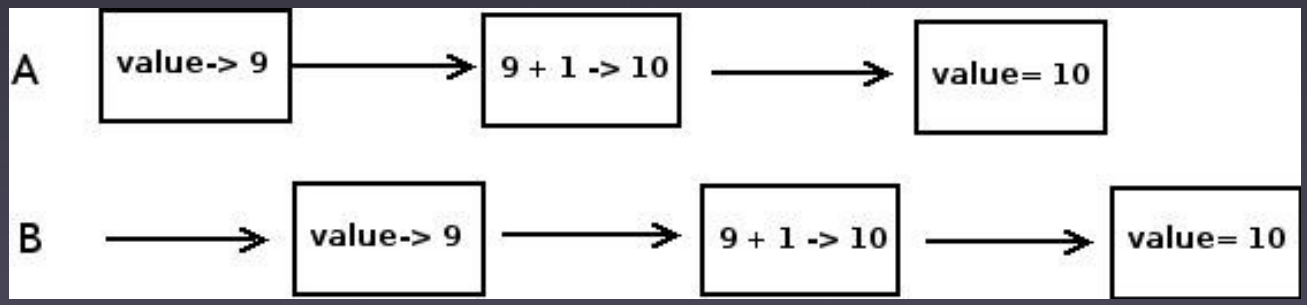
  3)Performance hazards

# Risk of threads (2)

- Safety hazards
  - Lack of sufficient synchronization can lead to strange results due to the unpredictable ordering of operations in multiple threads.

```
public class UnsafeSequence
{
    private int value;

    // Return a unique value
    public int getNext()
    {
        return value++;
    }
}
```




Its easy to forget that the ++-shorthand is actually 3 seperate operations (non-atomic)

# Risk of threads (3)

- Safety hazards
  - Previous example illustrated a so-called „race condition“.
  - To prevent this, access to the shared resource must be *synchronized*
  - Without this, the compiler, runtime (JVM) and hardware are free to play tricks with timing and ordering of actions to maximize performance

# Risk of threads (4)

- Liveness hazards
  - Bad implementation of multi-threaded applications can lead to „deadlocks“, causing the application to crash or loop forever.
  - Ex: If thread A waits for B to release a resource, but B never does, then A is unable to continue (*liveness failure*).
  - More on this in ch. 10

# Risk of threads (5)

- Performance hazards
  - While using threads give a net performance boost, it also adds overhead
  - Context switches (save current thread state, switch to another thread, restore previous state, etc) is expensive
  - The more threads, the more CPU time is spent scheduling threads instead of actually running them

# Threads are everywhere!

- All java programs use threads! Even if you never explicitly use threads in your code!
- JVM housekeeping tasks (garbage collection, finalization, etc). Swing/AWT for monitoring GUI-events. Servlets/RMI for handling client requests
- Dealing with concurrency is NOT optional, all developers must be aware of these issues

# Threads are everywhere! (2)

- When using frameworks, you automatically introduce concurrency in your application
- Since the framework is going to run your application code, your code must be thread-safe
- Ex: A Servlet can be called simultaneously from multiple threads (def. in spec.). Thus, servlet MUST be made thread safe. (Detailed servlet example later)

# Chapter 2 – Thread Safety

# Thread safety

- If multiple threads access the same mutable state variable without proper synchronization, your program is considered broken.
- Three ways to fix this:
  - *Don`t* share state variables across threads
  - Make the state variable *immutable*
  - Apply synchronization on all access to the state variable

# Thread-safe design

- The less code that has access to its state, the easier it is to make thread-safe objects (keep data private)
- Applying good object-oriented design principles (encapsulation, immutability, etc) helps when making thread-safe programs
- Its far easier to design a class to be thread-safe from the start, than to add thread-safety later on

# What is thread safety?

- Definition:
  „A class is *thread-safe* if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code.“

# A stateless object

- Ex: Stateless servlet:

```
@ThreadSafe
public class StatelessFactorizer extends Servlet {
        public void service(ServletRequest req, ServletResponse resp) {
                BigInteger i = extractFromRequest(req);
                BigInteger[] factors = factor(i);
                addResultToResponse(resp, factors);
        }
}
```

- Using only local variables, no shared data.
- Conclusion: Stateless objects are always thread-safe!

# Adding state

- Ex: A servlet with a „hit-counter“

```
@NotThreadSafe
public class UnsafeCountingFactorizer extends 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;
                addResultToResponse(resp, factors);
        }
}
```

- Problem: Hit counter is not accurate due to lack of synchronization.

# Lazy initialization issues

- Goal: Don`t init an expensive object untill its actually needed.

```
@NotThreadSafe
public class LazyInitRace {
        private ExpensiveObject instance = null;

        public ExpensiveObject getInstance() {
                if (instance == null)
                        instance = new ExpensiveObject();
                return instance;
        }
}
```

- „Check-then-act“ type of race condition
- Two threads might see instance is null and *two different instances* will be created (while everyone should actually get the same instance)

# Atomic operations

- To avoid race conditions, operations must be *atomic (indivisible)*
- The *compound action* ++*count* from prev. example was not atomic (but 3 operations)
- We can fix this by adding synchronization ourself....or use already existing thread-safe classes

# Built-in thread-safe objects

- The *java.util.concurrent*-package contains thread-safe atomic variable classes
- Ex. replace primitives like long, int, double with *Atomic*Long. *Atomic*Integer and *Atomic*Double.
- Where practical, use existing thread-safe classes instead of implementing your own.

# Adding more state

- Ex: Caching in our  factorizer servlet

```
@NotThreadSafe
public class UnsafeChachingFactorizer implements Servlet {
        private final AtomicReference<BigInteger> lastNumber
                = new AtomicReference<BigInteger>();
        private final AtomicReference<BigInteger[]> lastFactors
                = new AtomicReference<BigInteger[]>();

        public void service(ServletRequest req, ServletResponse resp) {
                BigInteger i = extractFromRequest(req);
                if(i.equals(lastNumber.get()) // Check
                        encodeIntoResponse(resp, lastFactors.get()); // Check
                else
                {
                        BigInteger[] factors = factor(i);
                        lastNumber.set(i);          // Write
                        lastFactors.set(factors);  // Write
                        encodeIntoResult(resp, factors);
                }
        }
}
```

- When updating one, you are *not* updating the other in the same atomic operation.

# Locking

- Java provides a built-in locking mechanism: the *synchronized* block

```
synchronized (lock) {
    // Access or modify shared state guarded by lock
}
```

- Two parts: Object to act as the lock, and block of code guarded by the lock
- Only one thread at a time can execute a block of code guared by a given lock.
- Syncronized blocks guarded by the same lock executes atomically with respect to one another.

# Reentrancy

- A thread blocks when waiting for another thread to release a lock
- Reentrancy: if a thread tries to aquire a lock *it already holds*, it succeeds

```
public class Widget {
        public synchronized void doSomething() {
                // Do stuff..
        }
}

public class LoggingWidget extends Widget {
        @Override
        public synchronized void doSomething() {
                // Do stuff..
                super.doSomething(); // Call method in superclass
        }
}
```

- Without reentrancy, deadlocks would occur

# Guarding state with locks

- When using locks to controll access to mutable state variables, *all* access must be guarded by the *same* lock.
- Having multiple locks for the same mutable state variable breaks the system.
- Easy for code maintainers to forget to add synchronization when, ex. adding a new method to a class. Make it clear for maintainers which lock to use!

# Liveness & performance

- Too much synchronization can affect performance, while too little can affect safety
- Having time-consuming operations (ex. I/O) inside a synchronized block should be avoided
- Always resist the temptation of sacrificing safety for simplicity  and performance

# Ex: Bad performance

- Synchr. the service method makes requests „queue up“ (handle one request at a time).

```
@ThreadSafe
public class UnsafeChachingFactorizer implements Servlet {
        @GuardedBy("this") private BigInteger lastNumber;
        @GuardedBy("this") private BigInteger[] lastFactors;

        public synchronized void service(ServletRequest req, ServletResponse resp)
                BigInteger i = extractFromRequest(req);
                if(i.equals(lastNumber)
                        encodeIntoResponse(resp, lastFactors);
                else
                {
                        BigInteger[] factors = factor(i);
                        lastNumber = i;
                        lastFactors = factors;
                        encodeIntoResult(resp, factors);
                }
        }
}
```

# Ex: Narrowing scope of sync. block

```
@ThreadSafe
public class UnsafeChachingFactorizer implements Servlet {
        @GuardedBy("this") private BigInteger lastNumber;
        @GuardedBy("this") private BigInteger[] lastFactors;
        @GuardedBy("this") private long hits;
        @GuardedBy("this") private long cacheHits;

        public synchronized long getHits() { return hits; }
        public synchronized double getCacheHitRatio() {
                return (double) cacheHits / (double) hits;
        }

        public void service(ServletRequest req, ServletResponse resp) {
                BigInteger i = extractFromRequest(req);
                BigInteger[] factors = null;;
                synchronized (this) {
                        ++hits;
                        if(i.equals(lastNumber) {
                                ++chacheHits;
                                factors = lastFactors.clone();
                        }
                }
                if (factors == null) {
                        factors = factor(i);
                        synchronized (this) {
                                lastNumber = i;
                                lastFactors = factors.clone();
                        }
                }
                encodeIntoResponse(resp, lastFactors);
        }
}
```

# Chapter 3 – Sharing objects

- 3.1: Visibility
- 3.2: Publication and escape
- 3.3: Thread confinement
- 3.4: Immutability
- 3.5: Safe publication

# Visibility (1)

- What happens here?

```
3  public class BadVisibility {
4
5      private static boolean ready;
6      private static int number;
7
8      private static class ReaderThread extends Thread {
9          public void run(){
10             while(!ready)
11                 Thread.yield();
12             System.out.println("Number is: " + number);
13         } // end run method
14     } // end ReaderThread class body
15
16     public static void main(String[] args){
17         new ReaderThread().start();
18         number = 42;
19         ready = true;
20     }
21 }
```

# Visibility (2)

- Alternatives:
  1. Prints out ”The number is 42” and terminates
  2. Prints out ”The number is 0” and terminates
  3. Runs in an eternal loop
- Answer:
  - You never know!
  - This is because of *caching*, and because the processor takes liberties to *reorder operations from different threads* unless we tell it to do differently

# Visibility (3)

- This program is about as simple as it gets when it comes to synchronisation
  - Two threads, two shared variables
- Even now, it is very hard to foresee what will happen when we run the program
- What if there were a hundred threads and many shared variables?

- Lesson: *Always use proper synchronisation when data is shared across threads!*

# Stale data

- *Stale data*: Data that are somehow ”out of date” – they have been updated after you read them, and are therefore incorrect
- This can be avoided by always synchronising methods that mutate the state of the object – ie the value of fields

# Non-atomic 64-bit operations

- *Out-of-thin-air safety*: A guarantee that even though a thread reads a variable without synchronisation, the stale value was placed there by a thread, it is not random
- This safety applies to all variables, except 64-bits variables (double and long)
- This is because 64 bit variables can be read in two operations with 32 bits each

# Locking and visibility

- Locking is used to guarantee that variable visibility is preserved correctly
- The compiler will not reorder instructions within a block of code that is synchronised
- Locking guarantees that if two threads, A and B, tries to access a block of code guarded by the same lock, and A gets to go first, all changes A did to variables is visible to B (no stale data)

# Volatile variables

- A field can be marked with the keyword **`volatile`**, which provides a weaker form of synchronisation upon the variable
  - Example: **`public volatile boolean done`**
- This implies that the compiler and runtime is told that any operations on the **`done`** variable should not be reordered or cached
- Volatile variables does not help if the updating of the variable requires more than one operation
- Should therefor only be used for variables where its operations are atomic: for instance setting boolean flags

- Rule: Locking guarantees both visibility and atomicity, volatile variables guarantees only visibility

# Rules of thumb for using volatile variables

- Use volatile variables only if **all** of these statements hold:
  - Writing to the variable does not depend on its current value, OR is is guaranteed that only a single thread updates the variable
  - The variable does not participate in invariants with other stateful variables
  - Locking when accessing the variable is not required for any other reason

- 3.1: Visibility
- 3.2: Publication and escape
- 3.3: Thread confinement
- 3.4: Immutability
- 3.5: Safe publication

# Publication and escape (1)

- *Publishing:* To make an object accessible to code outside of the current scope
- This can be done in many ways:
  - Storing a reference to it where other code can find it
  - Returning it from a non-private method
  - Passing it as a parameter to a method in another class
- This is not always desirable

# Publication and escape (2)

- Dangers of publication:
  - Publishing internal state variables compromises encapsulation and makes it difficult to preserve invariants
  - Publishing objects before they are fully constructed compromises thread safety
- *Escape*: An object that has been published unintendedly

# Publication and escape (3)

- There are multiple ways to let an object or state variable escape:
  - Storing a state variable in a **`public`** field
  - Having a public method that returns a state variable
  - Letting state variables contain inner classes (more on this later)
- *Alien method:* Methods from other classes, as well as reachable methods (neither private nor final) in the class itself
- Passing an object to an alien method has to be considered publishing that object

# Safe construction practices (1)

```
3  public class ThisEscape {
4
5      public ThisEscape(EventSource source){
6          source.registerListener(
7              new EventListener() {
8                  public void onEvent(Event e){
9                      doSomething(e);
10                 }
11             }
12         );
13     }
14 }
```

# Safe construction practices (2)

- When the inner class is published, the wrapping class is also published
- The problem is that the constructor has not yet finished running
- You are in fact publishing an object that is not properly constructed (uh-oh!)

- Rule: Never allow the `this`-reference to excape from the thread during construction of the object!

# Safe construction practices (3)

- A safe way to do this:

```
3  class SafeListener {
4      private final EventListener listener;
5
6      private SafeListener() {
7          listener = new EventListener() {
8              public void onEvent(Event e){
9                  doSomething(e);
10             }
11         };
12     }
13
14     public static SafeListener newInstance(EventSource source) {
15         SafeListener safe = new SafeListener();
16         source.registerListener(safe.listener);
17         return safe;
18     }
19 }
```

- 3.1: Visibility
- 3.2: Publication and escape
- 3.3: Thread confinement
- 3.4: Immutability
- 3.5: Safe publication

# Thread confinement

- Accessing shared, mutable data requires synchronisation
- The easiest way to avoid this is simply not to share data between threads
- This is called *thread confinement*
- Examples:
  - Swing framework
  - Connection pooling
- The Java language in itself has no means for defining that an object is confined to a thread – this is the responisibity of the programmer to ensure

# Ad-hoc thread confinement

- The simplest form of thread confinement
- The case when the responsibility for confinements relies entirely upon the implementation (no language ”help” used)
- Very fragile – easy to break by mistake
- The simplicity of making certain subsystems single-threaded can sometimes outweigh the fragility concerns of ad-hoc confinement
- Should be used sparingly

# Stack confinement (1)

- *Stack confinement:* An object can only be reached through local variables
- Simpler to maintain and less fragile than ad-hoc confinement
- Primitive type variables are always stack confined (due to value passing instead of reference passing)
- We still have to be careful: The language doesn’t enforce confinement
- Always document assumptions that certain objects are supposed to be used only within the current thread

# Stack confinement (2)

```
9  public class TheArk {
10
11     public int loadTheArk(Collection<Animal> candidates){
12
13         SortedSet<Animal> animals;
14         Ark ark = new Ark();
15         int numPairs = 0;
16         Animal candidate = null;
17
18         // animals confined to method, don't let them escape!
19         animals = new TreeSet<Animal>(new SpeciesGenderComparator<Animal>());
20         animals.addAll(candidates);
21         for(Animal a : animals){
22             if(candidate == null || !candidate.isPotentialMate(a)) {
23                 candidate = a;
24             } else{
25                 ark.load(new AnimalPair(candidate, a));
26                 ++numPairs;
27                 candidate = null;
28             }
29         }
30
31         return numPairs;
32     }
33
34 }
```

# ThreadLocals (1)

- **`ThreadLocal`** is a class used to contain separate values for each thread
- Think of it as a mutable singleton object local to each thread
- Provide three significant methods: **`initialValue()`**, **`get()`** and **`set(T value)`**.
- Use ThreadLocals with care; They should not be used for global variables or ”hidden” parameters to a method – this affects maintainability

# ThreadLocals (2)

- (example, see **`ThreadLocals.java`** in attached zip file)

- 3.1: Visibility
- 3.2: Publication and escape
- 3.3: Thread confinement
- 3.4: Immutability
- 3.5: Safe publication

# Immutability (1)

- Another way to escape synchronisation is to only use immutable objects (ie objects that does not change their state after creation)
- Immutable objects
  - Can not have stale values
  - Does not have visibility issues
  - Does not have to care about atomicity
  - Does not have to worry about escaping
- Rule: Immutable objects are always thread-safe

# Immutability (2)

- Declaring all fields in an object as **`final`** is not enough to make it immutable, since final fields can hold references to mutable objects
- An object is defined as mutable if all these three properties hold:
  - Its state cannot be modified after construction
  - All of its fields are **`final`**
  - It is properly constructed (the **`this`** reference does not escape during construction)

# Final fields

- The **`final`** keyword supports the construction of immutable objects
- If an object cannot be made totally immutable, it helps that at least *most* of the object is immutable by using **`final`** fields

- Rule: Make all fields **`final`** unless they need to be mutable

# Using volatility to publish immutable objects

- (example, see **`OneValueCache.java`** and **`VolatileCachedFactorizer`** in attached zip file)

- 3.1: Visibility
- 3.2: Publication and escape
- 3.3: Thread confinement
- 3.4: Immutability
- 3.5: Safe publication

# Safe publication (1)

- Sometimes it is necessary to publish objects
- It is important to consider how to do this in a thread-safe manner

# Safe publication (2)

```
3  public class HolderClient {
4
5      public Holder holder;
6      public void initialise() { holder = new Holder(42);  }
7  }
8
9  class Holder {
10     private int n;
11     public Holder(int n){     this.n = n;    }
12     public void assertSanity(){
13         if(n != n) throw new AssertionError("WTF?");
14     }
15 }
16
```

# Safe publication (3)

- What could possibly go wrong in this case?
  - A different thread than the publishing thread could retrieve a stale value of the Holder object (ie a **`null`** value even when the publishing thread has called the **`initialize()`**-method
    - This is, of course, due to visibility issues
  - A different thread than the publishing thread could get the right reference to the Holder object, but a stale value for the field in Holder
    - This is because even though it may seem like **`this.n = n;`** is the only statement here, every class is a subclass of **`Object`**, and the constructor of Object is run first

# Safe publication (4)

- The **`assertSanity()`**-method could actually fail, causing an **`AssertingError`**
  - The statement **`if(n != n)`** `...` is actually not atomic. The variable n is read twice. It is entirely possible to read a stale value the first time and the actual value the other time, causing the assertion to fail

# Safe publication (5)

- ”Now, this is utterly confusing! Why does this happen?!”
  - Publishing an object by storing its reference in a **`public`** field is unsafe
  - It is not the Holder object itself that is ”dangerous” here, but the way it is published

# Immutable objects and initialization safety

- The Java Memory Model (JMM) offers a special guarantee called *initialization safety* for sharing immutable objects
- This means that if you use immutable objects, you can access them even when you haven’t used synchronization to publish its refererence
- This means that if we had made the n field in the Holder class **`final`**, the publication through a **`public`** field would have been safe

# Safe publication best practices

- To publish an object safely, both the reference to the object and the object's state must be made visible to other threads *at the same time*
- A properly constructed object can be safely published by:
  - Initializing an object reference from a static initializer
  - Storing a reference to it into a **`volatile`** field or **`AtomicReference`**
  - Storing a reference to it into a **`final`** field of a properly constructed object
  - Storing a reference to it into a field that is properly guarded by a lock

# Safe publication in collections

- The last best practice also applies to certain types of collections:
  - Hashtable
  - synchronizedList, synchronizedSet, synchronizedMap
  - ConcurrentMap
  - Vector
  - CopyOnWriteArrayList and CopyOnWriteArraySet
  - synchronizedList, synchronizedSet
  - BlockingQueue, ConcurrentQueue
- The easiest, safest way to publish objects is to use a static initializer:
  - `public static Holder holder = new Holder(42);`

# Effectively immutable objects

- *Effectively immutable:* Objects that aren’t technically immutable, but whose state is never changed after publication
- These objects can be used safely by any thread without additional synchronization

# Object mutability rules

- Immutable objects can be published through any mechanism
- Effectively immutable objects must be safely published
- Mutable objects must be safely published, and must be either thread-safe or have their state guarded by a lock

# Object sharing policies

- To enable others to reason about the thread safety of your program, define the policy used, and document it
- The most common policies are:
  - Thread-confined
  - Shared read-only
  - Shared thread-safe
  - Guarded

# Chapter 4 – Composing objects

- 4.1: Designing a thread-safe class
- 4.2: Instance confinement
- 4.3: Delegating thread safety
- 4.4: Adding functionality to existing threadsafe classes
- 4.5: Documenting synchronisation policies

# Designing a thread-safe class (1)

- It is theoretically possible to write a thread-safe program that stores all its state in public static fields, but it is much harder to verify that the program is actually thread-safe
- The design process for a thread-safe class should include these three basic elements:
  - Identify the variables that composes the object's state
  - Identify the invariants that constrain the state variables
  - Establish a policy for managing concurrent access to the object's state

# Designing a thread-safe class (2)

- An object's state is composed of the values of its primitive typed fields, and the state of the objects it is composed of
- *Synchronization policy:* How an object coordinates access to its state without violating its invariants or postconditions
  - What combination of immutability, thread confinement and locking that is used to maintain thread safety

# Gathering synchronization requirements (1)

- Another way to define a thread-safe class is a class whose invariants hold also under concurrent access
- *State space:* The range of possible states an object can take on
- Some classes has invariants that restricts the state space
  - Example: The state space of int ranger from **`Integer.MIN_VALUE`** to **`Integer.MAX_VALUE`**, but a counter will restrict the values to be only positive

# Gathering synchronization requirements (2)

- Some operations may also have postconditions that restricts the state space
  - Example: For a counter in the state of 17, the only valid next state is 18
- If a class has invalid states, access to these state variables has to be encapsulated
- Also, if an operation can temporarily put an object in an inconistent state, it must be made atomic
- Rule: You cannot ensure thread safety without understanding an object's invariants and postconditions. Constraints on valid values or state transitions for state variables can create atomicity and encapsulation requirements

# State-dependent operations (1)

- Some objects can have operations with state-based preconditions
  - Example: It is impossible to remove an object from an empty queue; the queue most be in a non-empty state in order for the operation to be successful
- In single-threaded programs, an operation must fail if a precondition does not hold
- This is not the case with multithreaded programs, which have the choice of waiting until another thread has satified the preconditions (for instance put a new object into the queue) This is called *blocking*.

# State-dependent operations (2)

- Java's built-in mechanisms for blocking are **`wait`** and **`notify`**.
- These are often hard to use correctly, and it is often better to use library classes to get the desired behavior (for instance B**`lockingQueue`**)

# State ownership

- When we look at an object's state, we want to consider only the data it owns
- A class does usally not own objects passed to it as method parameters or constructor arguments, unless it is clearly stated that ownership is transferred to the class upon calling
- Collection classes often uses a kind of "shared ownership", where the collection class owns the state of the collection infrastructure, while the client code owns the objects contained in the collection

- 4.1: Designing a thread-safe class
- 4.2: Instance confinement
- 4.3: Delegating thread safety
- 4.4: Adding functionality to existing threadsafe classes
- 4.5: Documenting synchronisation policies

# Instance confinement (1)

- An object does not need to be thread-safe in order to be used in a multithreaded program
- A way to maintain thread safety with not thread-safe objects is to use instance confinement
- Instance confinement: to confine an object that is not thread-safe within another object, and make all methods accessing the unsafe object threadsafe
- This only works if the unsafe object never is allowed to escape

# Instance confinement (2)

- (see **`PersonSet.java`** in attached zip file for example)

# Instance confinement (3)

- Rule: Encapsulating data within an object confines access to the object's methods, making it easier to ensure that the data is always accessed using the right lock
- Instance confinement is one of the easiest way to build tread-safe classes
- It is used in for instance in **`Collections.synchronizedList`**, that only returns an instance confined version of a list object

# Java monitor pattern (1)

- *Java monitor pattern*: encapsulates all mutable state of an objects and guards it with its own intrinsic lock (ie by using the object itself as the lock)
- (see **`Counter.java`** for an example)

# Java monitor pattern (2)

- A variant of the Java monitor pattern is to use a private lock:
- See **`PrivateLock.java`** for an example
- Note that since the lock is private, no one else can obtain it and thus participate in the object's synchronization policy
- It is much easier to verify that a lock is properly used if it's private to the class than if it is public to all, as is the case with the Java monitor pattern

# Java monitor pattern (3)

- Example: Vehicle Tracker
- We have multiple vehicles (cars, trucks, buses etc) that we need to keep track of
- A vehicle is represented by a string identifier and a position
- We will have multiple threads reading and updating, so we need to support concurrent access to the information
- See **`MonitorVehicleTracker.java`** and **`MutablePoint.java`** for an example

- 4.1: Designing a thread-safe class
- 4.2: Instance confinement
- 4.3: Delegating thread safety
- 4.4: Adding functionality to existing threadsafe classes
- 4.5: Documenting synchronisation policies

# Delegating thread safety (1)

- The Java monitor pattern is useful when we are building classes from scratch or are composing classes out of components that are not thread-safe
- What if we are composing classes out of already thread-safe objects?
- Sometimes, classes composed from thread-safe are not thread-safe
- *Delegation*: To delegate thread-safety responsibilities to underlying classes

# Delegating thread safety (2)

- See **`DelegatingVehicleTracker.java`** and **`Point.java`** for an example

# When delegation fails

- Delegation does not work when a class has invariants that contains more state variables
- In this case, delegation is not enough, and additional precautions must be taken in order to ensure thread safety
- Rule: if a class is composed of multiple independent thread-safe state variables and has no operations that has any invalid state transitions, then it can safely delegate thread safety to the underlying state variables

# Publishing underlying state variables (1)

- In some cases, you can delegate thread safety to a class's underlying state variable and still be able to publish them
- Rule: if a state variable is thread-safe, does not participate in any invariants that constrain its value and has no prohibited state transitions for any of its operations, then it can be safely published

# Publishing underlying state variables (2)

- See **`PublishingVehicleTracker.java`** and **`SafePoint.java`** for an example

- 4.1: Designing a thread-safe class
- 4.2: Instance confinement
- 4.3: Delegating thread safety
- 4.4: Adding functionality to existing threadsafe classes
- 4.5: Documenting synchronisation policies

# Adding functionality to existing thread-safe classes

- The Java library provides lots of useful classes that provides often-needed functionality
- It is preferrable to use on of these before implementing a new class
- Sometimes, we have a class that *almost* fits our needs, but we have to add some functionality
- The best way to do this would be to modify the original class, but in most cases, this is not a viable alternative
- Another way to do this is to extend the original class and add the functionality you need
- This is a more fragile approach, as responsibility for the synchronization policy is now distributed over multiple source files

# Client-side locking (1)

- Many classes can, for some reason, not be subclassed
- We can then extend functionality by adding a "helper class"

# Client-side locking (2)

- See **`ListHelpers.java`** for an example

# Client-side locking (3)

- Why is BadListHelper unsafe?
- The ArrayList and the ListHelper does not synchronize on the same lock!
- Declaring a method as synchronized means that the class itself is used as the lock
- The ArrayList is using its own intrinsic lock
- This is a false synchronization guarantee

# Client-side locking (3)

- See **`ListHelpers.java`** for an example

# Composition

- The best alternative to add new operations to existing classes is composition
- This is guaranteed to work as long as the composer class holds the only reference to the list object
- See **`ImprovedList.java`** for an example

- 4.1: Designing a thread-safe class
- 4.2: Instance confinement
- 4.3: Delegating thread safety
- 4.4: Adding functionality to existing threadsafe classes
- 4.5: Documenting synchronisation policies

# Documenting synchronization policies

- Documentation is one of the most powerful and underutilized tools to maintain tread-safety
- Rule: Document a class's thread safety guarantees for its clients; document its synchronization policy for its maintainers

# Documenting synchronization policies (1)

- Every use of the tecniques described earlier is the result of careful thinking about synchronization policies. This has to be documented so that someone does not break it in the future
- Because thread-safety is so under-documented even in the Java library classes, sometimes you have to guess whether a class is thread-safe or not
- If a class is written for multi-threaded environments and it seems unreasonable that it could behave correctly while still being thread-safe (for instance ServletContext or HttpSession), you have to assume that these have been made thread-safe