Thread Safety

What it means:

- **'behaves correctly'**: satisfying its specification & preserving its rep invariant
(An invariant is a property that is always true of an ADT object instance, for the lifetime of the object)
- **'regardless of how threads are executed'**
- **'without additional synchronisation'**: the data type can't put preconditions on its caller related to timing (eg. you can't call get() while set() is in progress)

Strategy 1: Confinement

- Thread confinement is a simple idea: you avoid races on **mutable** data by keeping that data confined to a single thread. Don't give any other threads the ability to read or write the data directly.
- **Local variables** are always thread confined. A local variable is stored in the stack, and each thread has its own stack.
- But be careful – the variable is thread confined, but if it's an **object reference**, you also need to check the object it points to. If the object is mutable, then we want to check that the object is confined as well – there can't be references to it that are reachable from any other thread.
- **Avoid Global variables**: Unlike local variables, static variables are not automatically thread confined.

Strategy 2: Immutability

- Achieve thread safety by using **immutable references and data types**. Immutability tackles the shared-mutable-data cause of a race condition and solves it simply by making the shared data not mutable.
- **Final** variables are immutable references, so a variable declared final is safe to access from multiple threads. You can only read the variable, not write it. *Be careful, because this safety applies only to the variable itself, and we still have to argue that the object the variable points to is immutable.*
- *The final modifier is applicable for variable but not for objects, Whereas immutability applicable for an object but not for variables. By declaring a reference variable as final, we won't get any immutability nature, even though reference variable is final. We can perform any type of change in the corresponding Object. But we can't perform reassignment for that variable. Eg. Hash map*
- But we need a **stronger definition** of immutability because we still allow the data type the freedom to mutate its rep (as long as mutations are invisible to clients). So to be confident that an immutable data type is threadsafe without locks:

    1. No mutator methods
    2. All fields are **private and final**
    3. No mutation whatsoever of mutable objects in the rep (not even beneficent mutation)
    4. No rep exposure

Strategy 3: Using Thread-safe Data Types

- Achieve thread safety by storing **shared mutable** data in **existing thread-safe data** types.
- Common in **Java API to find 2 mutable data types** that do the same thing, one thread-safe and the other not. E.g. `StringBuffer` (thread-safe) vs `StringBuilder`. Thread-safe data types usually incur a **performance penalty** compared to an unsafe type.

Strategy 4: Synchronisation

Use synchronization to keep the threads from accessing the variable **at the same time**. Synchronization is what you need to build your own thread-safe data type.

Since race conditions caused by concurrent manipulation of shared mutable data are disastrous bugs — **hard to discover, hard to reproduce, hard to debug** — we need a way for concurrent modules that share memory to **synchronize** with each other.

- **Intrinsic Lock:** A thread that needs exclusive access to an object's fields has to **acquire** the object's lock before accessing them, and then **releasing** the intrinsic lock once it is done. Other threads trying to access the object will **block** until the thread holding the lock releases it.

- **Synchronized Methods:** When a thread invokes a synchronized method, it acquires the intrinsic lock for that method's object and releases it **when the methods returns**. The lock is released even if the method returns due to **an uncaught exception**. If done in **a static** method, the thread acquires the lock for the **class object associated with the class**.

- **Synchronized Statements**: must specify the object that provides the intrinsic lock. Synchronizing over separated lock objects can provide fields synchronization, without forcing synchronization between methods calls.
  > If lock is static (private static final Object lock), all instances of MyClass will share that lock
  > If lock is non-static (private final Object lock), each instance of the class will have its own lock, so only calls of the method on the same object will lock each other.

- **Guarded Blocks**: Guarded blocks are part of every Java object, and can be constructed using the `wait(), notify()` and `notifyAll()` methods.
 > The `wait` method suspend the current thread. When a thread invokes wait, it must own the object's intrinsic lock, that is why calls to wait are usually wrapped in a synchronized method or statement. The invocation of the wait method suspends the thread execution and releases the lock.
 > At some point, another thread will acquire the object's intrinsic lock and invoke `notifyAll` to inform all threads waiting that something important has happened. After the second thread has released the lock, the waiting threads will reacquire the lock and resume execution by returning from the wait invocation.
 > `Notify` wakes up a single thread. The concrete thread that is woken up cannot be specified, therefore, it is useful only if we do not care which thread is woken up.

Deadlock
When used properly and carefully, locks can prevent race conditions. But then another problem rears its ugly head. Because the use of locks requires threads to wait (`acquire` blocks when another thread is holding the lock), it's possible to get into a situation where two threads are waiting for each other — and hence neither can make progress.

Building Thread-safe Classes
In general, no sharing ⇒ Thread safety and sharing constants only ⇒ thread safety.
Otherwise, make sure every shared variable is of a type which is thread-safe.

Strategy 1: From Scratch
Design process include these 4 steps:

1. Identify the variables that form the object's state
   - State includes all its mutable variables.
   - if of primitive type, the fields comprise the entire state
   - If object has fields that are references to other objects, its state encompass fields from those as well.
   - Tip: use 'final' if you can
2. Identify the requirements (e.g. invariants, post-conditions) that constrain the state variables
   - Constraints on the valid values or state transitions for state variables can **create atomicity and encapsulation requirements**
3. Establish a policy for managing concurrent access to the object's state
   - Update related state variables in a **single atomic operation**
   - For each mutable variable that may be accessed by more than one thread, all assesses to that variable must be performed with the **same lock** held.
   - Every shared, mutable variable should be guarded by **exactly one lock**. Make it clear to maintainers which lock that is.
   - For every invariant that involves more than one variable, all the variables involved in that invariant must be **guarded by the same lock.**

4. Implement the policy
   - Make sure every access of any variable is guarded by the lock according to the policy.
   - Make sure access of the related variables in the same method is in synchronized block.
   - Add waiting (and notify) to handle pre-conditions.

Strategy 2: By Extending the Class

You build a thread-safe class through extending the class if:
• there is a thread-safe class supporting almost all the operations you want;
• and you know the locking policy of the class (so that you can follow the same policy);
• and you don't have the source code.

Private Lock (`private final Object myLock = new Object();`)
+ Making the lock object private encapsulates the lock so that client code cannot acquire it, whereas a publicly accessible lock allows client code to participate in its synchronisation policy-correctly or incorrectly.
- In PrivateLockExample, we have a synchronized block guarded by a private lock in `executeTask()` method. If one thread enters `executeTask()` and acquire the lock, any other threads entering other methods within this class guarded by the same lock will have to wait in order to acquire it. When we extend this class to add more methods, these methods also need to be synchronized because they need to use the same shared data. Since the lock is private in the base class, the extended class won't have access to it. Thus, the threads **can interleave** between `executeTask()` and the new methods guarded by new locks.
+/- Private locks are **flexible and risky**
+/- More fragile than modifying the class directly, because the implementation of the synchronization policy is now **distributed over multiple**, separately maintained source files.

Strategy 3: Through Client-side Locking

You can build a thread-safe class through client-side locking if:
• We don't have the source code or enough access to the internal state so that we can extend the class.
• You know the client's locking policy (i.e., there are no private locks).

Even more fragile than extending the class because it distributes the locking policy for a class into classes that are totally unrelated.
– Modifying the class: keeps the locking code in the same class
– Extending the class: distributes the locking code in the class hierarchy

Strategy 4: Through Composition

Create a new class and use `synchronized` on all the methods available to `List` -> `improvedList`
• It is less fragile if list is accessed only through improvedList.
• It doesn't care whether list is thread-safe or not.
• It adds **performance penalty** due to the extra layer of synchronization. Example: ImprovedList.java

The above methods assume that the only way to access the state in the class is through calls of visible methods. We also need:

Instance Confinement

- Mentioned right at the start
- Identify the intended scope of objects
  – To a class instance (e.g., a private data field)
  – To a lexical scope (e.g., a local variable)
  – To a thread (e.g., an object which is not supposed to be shared across threads)
- Instance confinement is one of the easiest ways to build thread-safe classes

When is it safe to publish state variables?

– If a state variable is thread-safe, does not participate in any invariant that constrain its value, and has no prohibited state transitions for any of its operations, then it can be safely published.

– It still might not be a good idea, since publishing mutable variables constrains future development and opportunities for sub-classing.

Delegating Thread Safety

• If a class is composed to multiple **independent** thread-safe state variables, then it can delegate thread safety to the underlying state variables.
• If a class has invariants the relate its state variables, then delegation may not work.

Testing

• For sequential programs, – Finding the right inputs
• For concurrent programs, – Finding the right inputs and scheduling
– To be able to generate more scheduling, we could use `Thread.sleep()`, and synchronizers.

Synchronisers

A synchronizer is an object that coordinates the control flow of threads based on its state.

1. Semaphores

A semaphore maintains a set of permits. Each `acquire()` blocks if necessary until a permit is available, and then takes it. Each `release()` adds a permit, potentially releasing a blocked acquirer.

2. Cyclic Barriers

A synchronization aid that allows a set of threads to all wait for each other to reach **a common barrier point**. The barrier is often called cyclic because it can be re-used after the waiting threads are released.
- Allow a **fixed** number of parties to gather at a barrier point
`.await()` -> blocks until all parties arrive. After all threads arrived, it continues.

3. Countdown Latch

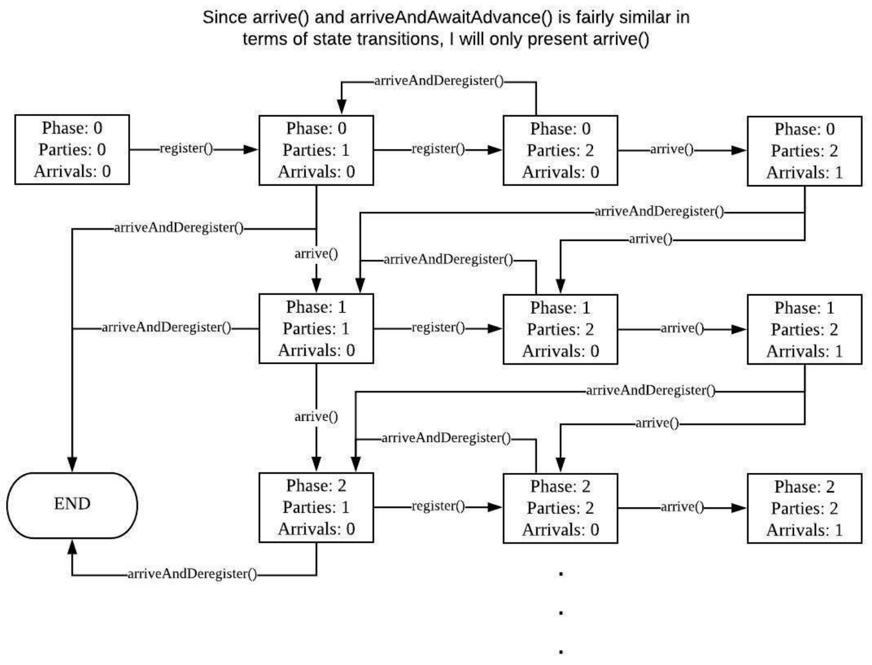
A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.
- Initialised to **number of events** to wait for (positive number)

```
final CountDownLatch start = new CountDownLatch(1); // look at red book
```

- `.countDown()` -> decrement counter
- `.await()` -> blocks until count reaches zero

Latch: single-use object. Once latch enters terminal state, it cannot be reset. Used for waiting for events.
Barrier: all threads must come together at a barrier point at the same time in order to proceed. Waiting for threads.

4. Phaser


Since arrive() and arriveAndAwaitAdvance() is fairly similar in terms of state transitions, I will only present arrive()
Phase: 0 Parties: 0 Arrivals: 0
register()
Phase: 0 Parties: 1 Arrivals: 0
register()
Phase: 0 Parties: 2 Arrivals: 0
arrive()
Phase: 0 Parties: 2 Arrivals: 1
arriveAndDeregister()
arrive()
Phase: 1 Parties: 1 Arrivals: 0
Phase: 1 Parties: 2 Arrivals: 0
Phase: 1 Parties: 2 Arrivals: 1
Phase: 2 Parties: 1 Arrivals: 0
Phase: 2 Parties: 2 Arrivals: 0
Phase: 2 Parties: 2 Arrivals: 1
END

• Number of threads need not be known at Phaser creation time. They can be added dynamically.
• Can be reset and hence is, reusable.
• Allows threads to wait (method `arriveAndAwaitAdvance()`) or continue with its execution (method `arrive()`). `arriveAndDeregister()` record arrival, but don't block. `awaitAdvance()` blocks until all parties arrive.
• Supports multiple Phases.

Testing for Concurrency

Testing for Correctness

– Safety: nothing bad ever happens
– Liveness: something good eventually happens (e.g., no deadlock)

Testing for Performance

– Throughput: the rate at which a set of concurrent tasks is completed
– Responsiveness: the delay between a request and completion of some action

Step 1: Identifying Specification

You must know what is correct.
- Identify:
    > Class invariants which specify relationships among the variables;
    > Pre/post-conditions for each method;
    > Whether the class is thread-safe and how its states guarded

Step 2: Basic Unit Tests

• Create an object of the class, call its methods (in different sequences with different inputs) and assert post-conditions and invariants.

Step 3: Test for Concurrency

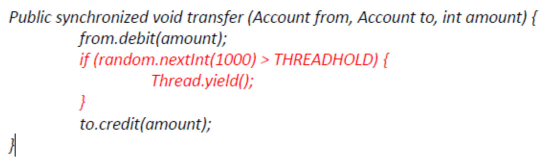
• Set up multiple threads performing operations over some amount of time and then somehow test that nothing went wrong
    - Mind that the test programs are concurrent programs too!
• It's best if checking the test property does not require any synchronization

Random

- Test data should be generated randomly.
- Random number generator can **create couplings** between classes and timing artifacts because most random number generator classes are **thread-safe** and therefore introduce **additional synchronization**. ** use pseudo-random number generator (using xorShift)

Generating More Scheduling

• Test with more active threads than CPUs
• Testing with different processor counts, operating systems, and processor architectures
• Encourage context switching using Thread.yield() or Thread.sleep(10)



```
Public synchronized void transfer (Account from, Account to, int amount) {
            from.debit(amount);
            if (random.nextInt(1000) > THREADHOLD) {
                        Thread.yield();
            }
            to.credit(amount);
}
```

Testing Blocking Operations

```
try {
  taker.start();
  Thread.sleep(LOCKUP_DETECT_TIMEOUT);
  taker.interrupt();                    // terminate
  taker.join(LOCKUP_DETECT_TIMEOUT);// join after timeout
  assertFalse(taker.isAlive());    // the taker should not be alive for some time
} catch (Exception unexpected) {
  assertTrue(false);                    // should not happen
}
```

Step 4: Testing for Performance

• Identify appropriate test scenarios –how the class is used
• Sizing empirically for various bounds, e.g., number of threads, buffer capabilities, etc.

Concurrency for Performance

Use an existing library if you can

Concurrency-Related Libraries:

Synchronizers

(previous pages)

Synchronized Collections

- Collections.synchronizedXxx
- Collections are essentially built with synchronized methods
- But **not ideal** because it is not efficient if list is big and/or method is slow (cause you synchronized (list) the whole method)
- *Look at Hidden Iterators* (it is hard to use locking everywhere a shared collection might be iterated)
- Alternative to locking a collection: Clone the collection, lock and iterate the copy

Concurrent Collections

- Replacing synchronized collections with concurrent collections can offer dramatic scalability improvement with little risk

<u>ConcurrentHashMap:</u>

> Hash map designed for concurrency. Uses a finer-grained locking mechanism called **lock striping** (uses an array of 16 locks, each lock guards 1/16 of the hash buckets, bucket N is guarded by lock N mod 16)
> Does **not** support **client-based locking** (16 locks are private objects in concurrenthashmap class itself. A client cannot access these locks, you can't lock based on class too!)
> Iterators returned by ConcurrentHashMap are **weakly consistent** (when you are modifying this part of datastructure, some other thread can modify other part of the datastructure.) instead of fail-fast

<u>CopyOnWriteArrayList</u>

> Concurrent replacement for a synchronized list that offers better concurrency in **some** common situations
> A **new copy** of the collection is created & published every time it is modified.
> **All write operations are protected** by the same lock & **read operations are not protected**
> Not efficient if most cases are writing, then overhead is copy

<u>BlockingQueue</u>

> Producer-Consumer patterns are very common in programs; Usually some kind of buffering is involved between P and C; The buffer can be implemented as a blocking queue.
> Blocking queues are a powerful resource management tool for building reliable applications: they make your program more robust to overload by throttling activities that threaten to produce more work than can handled.

FutureTask

> Made up of **Future** and **Callable** (result bearing relative of Runnable)
> Several ways to complete: Normal completion; cancellation; and exception
> Once a FutureTask is completed it **cannot be restarted**.
> `Future.get()` returns the result immediately if 'the future is here' (Task is completed) and Blocks if the task is not complete yet.

Executor and Thread Pools

- Most concurrent applications are organized around the execution of tasks: abstract, discrete units of work.
- Need to identify sensible task boundaries.
- Ideally, tasks are independent activities. Independence facilitates concurrency

One Thread Per Task

More responsive than executing individual client requests sequentially
• Task processing is offloaded from the main thread – more responsive.
• Tasks can be processed in parallel – improved throughput.
• Task-handling code must be thread-safe, because it may be invoked concurrently for multiple tasks.
Works under light or moderate load.

Unbounded Thread Creation

For production purposes (large webservers for instance) task-per-thread has some drawbacks:
– Thread creation and tear down involves the JVM and OS. For lots of lightweight threads this is **not very efficient**.
– Active Threads **consume extra memory**, for instance to provide for a thread stack.
– If there are less CPU's than threads, some threads sit idle, consuming memory.
– There is a **limit** on how many threads you can have concurrently. If you hit this limit your program will most likely become **unstable**.

Executor

Provides a standard means of decoupling task submission from task execution. (The Runnable is the task itself, the method execute defines how it is executed.)

Advantage of Thread Pools

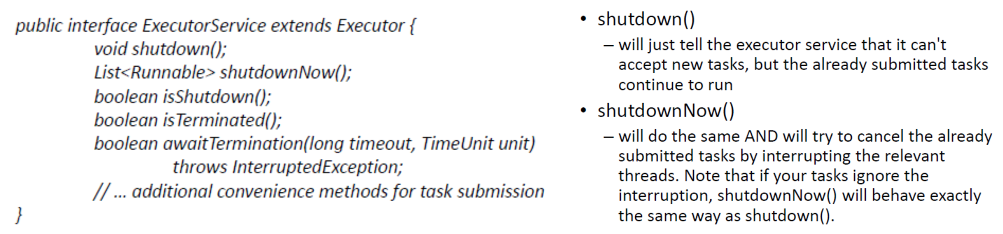
• Reusing an existing thread; reduce thread creation and teardown costs.
• No latency associated with thread creation; improves responsiveness.

By properly tuning the size of the thread pool, you can have enough threads to keep the processors busy while not having so many that your application runs out of memory or thrashes due to competition among threads for resources

Thread Pool Implementations

• `newFixedThreadPool:` Fixed-size thread pool; creates threads as tasks are submitted, up to the maximum pool size and then attempts to keep the pool size constant
• `newCachedThreadPool`: Boundless, but the pool shrinks and grows when demand dictates so
• `newSingleThreadExecutor:` A single worker thread to process tasks, sequentially according to the order imposed by the task queue
• `newScheduledThreadPool:` A fixed-size thread pool that supports delayed and periodic task execution.

Shut Down Executor



```
public interface ExecutorService extends Executor {
        void shutdown();
        List<Runnable> shutdownNow();
        boolean isShutdown();
        boolean isTerminated();
        boolean awaitTermination(long timeout, TimeUnit unit)
                throws InterruptedException;
        // ... additional convenience methods for task submission
}
```

• shutdown()
  – will just tell the executor service that it can't accept new tasks, but the already submitted tasks continue to run
• shutdownNow()
  – will do the same AND will try to cancel the already submitted tasks by interrupting the relevant threads. Note that if your tasks ignore the interruption, shutdownNow() will behave exactly the same way as shutdown().

Thread pools work best when tasks are homogeneous and independent.
– Dependency between tasks in the pool creates constraints on the execution policy which might result in problems (deadlock, liveness hazard, etc.)
– Long-running tasks may impair the responsiveness of the service managed by the Executor.
– Reusing threads create channels for communication between tasks – don't use them.

The ideal size for a thread pool depends on the types of tasks and the deployment system
– If it is too big, performance suffers
– If it is too small, throughput suffers

Code Smell

Does not exhibit bugs, but signs of bad code and may introduce bugs in the future

Repeated Code

Code fragments containing very similar code.
Solution: Refactor to use functions

Long Methods

The general rule of thumb is that the method should have a single, clear objective
Solution: Break up code into other methods and use these methods in your code.

Black Hole Classes

Classes start small. Adding more responsibility to a class attracts further responsibilities – Eventually the class behaves like a Black hole.

Data Class

Data classes are too small of a class without much responsibility. (only has attributes and getter, setter methods)
Solution: Add more responsibilities to the class to remove 'data class' smell

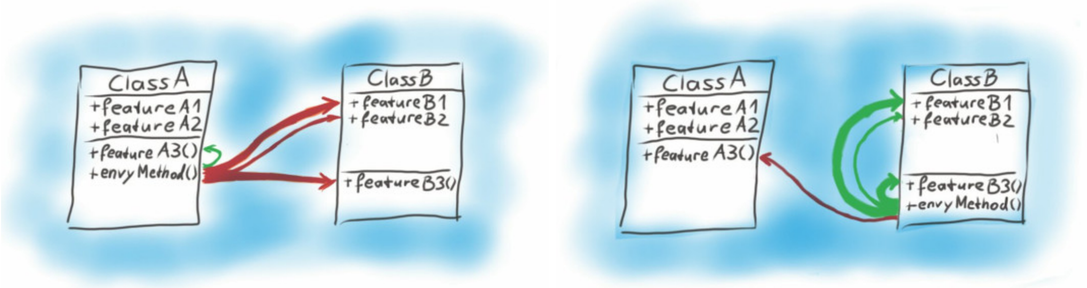
Data Clumps

Data clumps are opposite to data classes – It is better to create a class if a set of parameters are used repeatedly
EG. `public void doSomething1(int x, int y, int z) // create Point3D(x,y,z)`
Solution: Create a class for the parameters (and give it responsibilities to avoid 'data class' smell

Shotgun Surgery

Adding a simple feature may need to change all over the code

Feature Envy



A method has Feature Envy on another class, if it uses more features (i.e. fields and methods) of another class than of its own.
Solution: Similar to the refactoring for Data Classes, you have to move the methods to its preferred class. For the example above, you have to move `envyMethod()` to `ClassB`. (Diagram to right). We reduced the coupling (red arrows) between our classes and raised the cohesion (green arrows) inside our classes.

Long Message Chains

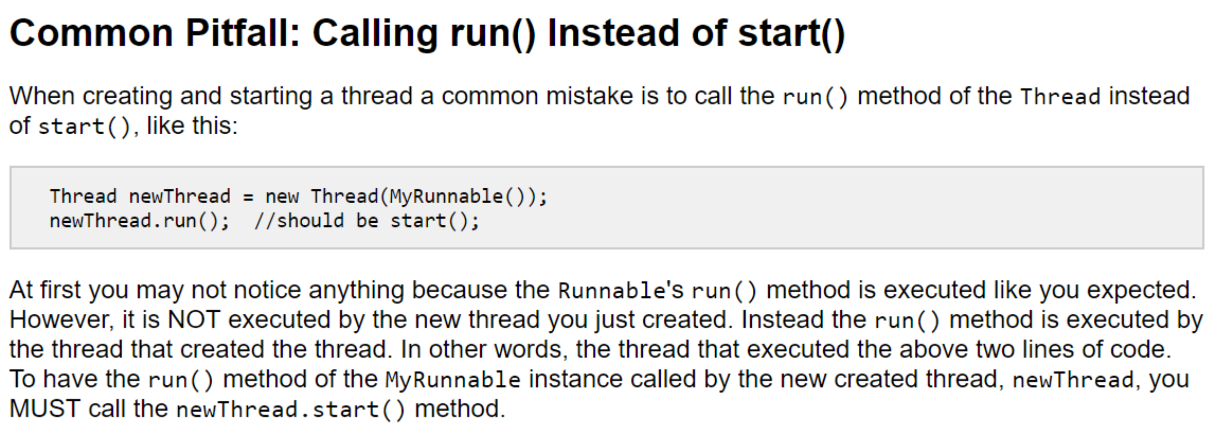
Should avoid calling along sequence of messages without checks – E.g. `A.B().C().D().E()`
A message chain couples a client of the method to the structure of the navigation. Any change to the intermediate relationships requires the client to have to change.
Solution: shorten chain by using Hide Delegate (or middle man)

Speculative Generality

There is an unused class, method, field or parameter. Should avoid over engineering based on unlikely generalisation. Concentrate on the features needed, throw away all other features.

Refused Bequest

Bad inheritance – inherit multitudes of unnecessary methods
Solution: If inheritance makes no sense and the subclass really does have nothing in common with the superclass, eliminate inheritance. Create a field and put a superclass object in it, delegate methods to the superclass object (Another way is to shift the common class into another abstract class, then inherit for one and extend for another).



Common Pitfall: Calling run() Instead of start()

When creating and starting a thread a common mistake is to call the `run()` method of the `Thread` instead of `start()`, like this:

```
Thread newThread = new Thread(MyRunnable());
newThread.run();  //should be start();
```

At first you may not notice anything because the `Runnable`'s `run()` method is executed like you expected. However, it is NOT executed by the new thread you just created. Instead the `run()` method is executed by the thread that created the thread. In other words, the thread that executed the above two lines of code. To have the `run()` method of the `MyRunnable` instance called by the new created thread, `newThread`, you MUST call the `newThread.start()` method.

Comments

- Used to document

– Application programmer interfaces (APIs)
– Choices of data structure and algorithms

- Don’ts

– Over comment
– Explain how the code works

Comments that reflect potential changes to be made in a different method are indication of code smells.
E.G. `/* if you change this function, then you need to change the "checkEmptyByString" function */`