Effective Concurrency for the Java™ Platform

Brian Goetz
Senior Staff Engineer
Sun Microsystems, Inc.
brian.goetz@sun.com

TS-2388
The Big Picture

Writing correct concurrent code is difficult, but not impossible

Using good object-oriented design techniques can make it easier
About the Speaker

- Brian Goetz has been a professional software developer for 20 years

- Author of *Java Concurrency in Practice*

- Author of over 75 articles on Java™ platform development

- Member of Java Community Process℠ (JCP℠) expert groups for JSRs 166 (Concurrency), 107 (Caching), and 305 (Safety annotations)

- Regular presenter at the JavaOne℠ conference, SDWest, OOPSLA, JavaPolis, and No Fluff, Just Stuff

*JSR = Java Specification Request*
Agenda

Introduction

Rules for Writing Thread-Safe Code
  Document Thread-Safety Intent and Implementation
  Encapsulate Data and Synchronization
  Prefer Immutable Objects
  Exploit Effective Immutability

Rules for Structuring Concurrent Applications
  Think Tasks, Not Threads
  Build Resource-Management Into Your Architecture
  Decouple Identification of Work from Execution

Rules for Improving Scalability
  Find and Eliminate the Serialization
Agenda

Introduction

Rules for Writing Thread-Safe Code
- Document Thread-Safety Intent and Implementation
- Encapsulate Data and Synchronization
- Prefer Immutable Objects
- Exploit Effective Immutability

Rules for Structuring Concurrent Applications
- Think Tasks, Not Threads
- Build Resource-Management Into Your Architecture
- Decouple Identification of Work from Execution

Rules for Improving Scalability
- Find and Eliminate the Serialization
Introduction

● This talk is about identifying patterns for concurrent code that are less fragile
  ● Conveniently, many are the good practices we already know
  ● Though sometimes we forget the basics
● Feel free to break (almost) all the rules here
  ● But be prepared to pay for it at maintenance time
  ● Remember the core language value: Reading code is more important than writing code
Agenda

Introduction

Rules for Writing Thread-Safe Code

- Document Thread-Safety Intent and Implementation
- Encapsulate Data and Synchronization
- Prefer Immutable Objects
- Exploit Effective Immutability

Rules for Structuring Concurrent Applications

- Think Tasks, Not Threads
- Build Resource-Management Into Your Architecture
- Decouple Identification of Work from Execution

Rules for Improving Scalability

- Find and Eliminate the Serialization
Document Thread-Safety

- One of the easiest way to write thread-safe classes is to build on existing thread-safe classes
  - But how do you know if a class is thread-safe?
    - The documentation should say, but frequently doesn't
  - Can be dangerous to guess
    - Should assume not thread-safe unless otherwise specified

- Document thread-safety design intent
  - Class annotations: @ThreadSafe, @NotThreadSafe
    ```java
    @ThreadSafe
    public class ConcurrentHashMap { .... }
    ```

- With class-level thread-safety annotations:
  - Clients will know whether the class is thread-safe
  - Maintainers will know what promises must be kept
  - Tools can help identify common mistakes
Document Thread-Safety

● Should also document **how** a class gets its thread-safety
  ● This is your **synchronization policy**

● The Rule:
  ● When writing a variable that might next be read by another thread, or reading a variable that might last have been written by another thread, **both** threads must synchronize using a common lock

● Leads to design rules of the form **hold lock L when accessing variable V**
  ● We say **V is guarded by L**

● **These rules form protocols for coordinating access to data**
  ● Such as “Only the one holding the conch shell can speak”

● **Only work if all participants follow the protocol**
  ● If one party cheats, everyone loses
Document Thread-Safety

- Use `@GuardedBy` to document your locking protocols
- Annotating a field with `@GuardedBy("this")` means:
  - Only access the field when holding the lock on “this”

```java
@ThreadSafe
public class PositiveInteger {
  // INVARIANT: value > 0
  @GuardedBy("this") private int value = 1;

  public synchronized int getValue() { return value; }
  public void setValue(int value) {
    if (value <= 0)
      throw new IllegalArgumentException(....);
    synchronized (this) {
      this.value = value;
    }
  }
}
```
- Simplifies maintenance and avoids common mistakes
- Like adding a new code path and forgetting to synchronize
- Improper maintenance is a big source of concurrency bugs
Document Thread-Safety

- For primitive variables, `@GuardedBy` is straightforward
- But what about
  ```java
  @GuardedBy("this") Set<Rock> knownRocks = new HashSet<Rock>();
  ```
- There are three different types of potentially mutable state
  - The `knownRocks` reference
  - The internal data structures in the `HashSet`
  - The elements of the collection
- Which types of state are we talking about? All of them?
- It varies, but we can often tell from context
  - Are the elements owned by the class, or by clients?
  - Are the elements thread-safe?
  - Is the reference to the collection mutable?
    ```java
    @GuardedBy("this") final Set<Rock> knownRocks = ....
    ```
Document Thread-Safety

- For complicated data structures, draw a diagram identifying ownership and synchronization policies
  - Color each state domain with its synchronization policy

```java
@ThreadSafe public class Rock { .... }
```

```java
@GuardedBy("this") final Set<Rock>
knownRocks = new HashSet<Rock>();
```

- Very effective for designing and reviewing code!
  - Frequently identifies gaps or inconsistencies in synchronization policies
Summary: Document Thread-Safety

- Document classes as `@ThreadSafe` or `@NotThreadSafe`
  - Saves your clients from guessing wrong
  - Puts maintainers on notice to preserve thread-safety
- Document synchronization policy with `@GuardedBy`
  - Helps you make sure you have a clear thread-safety strategy
  - Helps maintainers keep promises made to clients
  - Helps tools alert you to mistakes
- Use diagrams to verify thread-safety strategies for nontrivial data structures
- Inadequate documentation → fragility
Agenda

Introduction

Rules for Writing Thread-Safe Code
- Document Thread-Safety Intent and Implementation
- Encapsulate Data and Synchronization
- Prefer Immutable Objects
- Exploit Effective Immutability

Rules for Structuring Concurrent Applications
- Think Tasks, Not Threads
- Build Resource-Management Into Your Architecture
- Decouple Identification of Work from Execution

Rules for Improving Scalability
- Find and Eliminate the Serialization
Encapsulate Data and Synchronization

- Encapsulation promotes clear, maintainable code
  - Reduces scope of effect of code changes
- Encapsulation similarly promotes thread safety
  - Reduces how much code can access a variable
  - And therefore how much be examined to ensure that synchronization protocols are followed
- Thread safety is about *coordinating access to shared mutable data*
  - Shared—might be accessed by more than one thread
  - Mutable—might be modified by some thread
- Less code that accesses a variable means fewer opportunities for error
Encapsulate Data and Synchronization

- **Encapsulation makes it sensible to talk about individual classes being thread-safe**

- **A body of code is thread-safe if:**
  - *It is correct in a single-threaded environment, and*
  - *It continues to be correct when called from multiple threads*
    - Regardless of interleaving of execution by the runtime
    - Without additional coordination by callers

- **Correct means** **conforms to its specification**
  - Often framed in terms of *invariants and postconditions*
    - These are statements about *state*

- Can’t say a body of code guarantees an invariant unless no other code can modify the underlying state
  - Thread-safety can only describe a body of code that manages all access to its mutable state
  - Without encapsulation, that's the **whole program**
Encapsulate Data and Synchronization

- **Is this code correct? Is it thread-safe?**

  ```java
  public class PositiveInteger {
      // INVARIANT: value > 0
      @GuardedBy("this") public int value = 1;

      public synchronized int getValue() { return value; }

      public synchronized void setValue(int value) {
          if (value <= 0)
              throw new IllegalArgumentException(...);
          this.value = value;
      }
  }
  ```

- We can’t say unless we examine all the code that accesses `value`
  - Doesn’t even enforce invariants in single-threaded case
  - Difficult to reason about invariants when data can change at any time
  - Can’t ensure data is accessed with proper synchronization
Encapsulate Data and Synchronization

- **Without encapsulation, cannot determine thread-safety without reviewing the entire application**
  - Much easier to analyze one class than a whole program
  - Harder to accidentally break thread safety if data and synchronization are encapsulated

- **We can** build thread-safe code without encapsulation
  - But it's fragile
  - Requires code all over the program to follow the protocol

```java
public final static Object lock = new Object();
@GuardedBy("lock")
public final static Set<String> users = new HashSet<String>();
```

- Imposing locking requirements on external code is asking for trouble
  - *Fragility increases with the distance between declaration and use*
Encapsulate Data and Synchronization

- Sometimes we can push the encapsulation even deeper
  - Manage state using thread-safe objects or volatile variables
  - Even less fragile—can't forget to synchronize
  - *But only if class imposes no additional invariants*

- *Can transform this*
  ```java
  public class Users {
      @GuardedBy("this")
      private final Set<User> users = new HashSet<User>();
      
      public synchronized void addUser(User u) { users.add(u); }
      
  }
  ```

- *Into this*
  ```java
  public class Users {
      private final Set<User> users = Collections.synchronizedSet(new HashSet<User>());
      
      public void addUser(User u) { users.add(u); }
      
  }
  ```
Encapsulate Data and Synchronization

- If a class imposes invariants on its state, it must also provide its own synchronization to protect these invariants
  - Even if component classes are thread-safe!

- UserManager follows The Rule
  - But still might not be thread-safe!

```java
class UserManager {
    // Each known user is in exactly one of {active, inactive}
    private final Set<User> active = Collections.synchronizedSet(new HashSet<User>());
    private final Set<User> inactive = Collections.synchronizedSet(new HashSet<User>());

    // Constructor populates inactive set with known users
    public void activate(User u) {
        if (inactive.remove(u))
            active.add(u);
    }

    public boolean isKnownUser(User u) {
        return active.contains(u) || inactive.contains(u);
    }
}
```
Encapsulate Data and Synchronization

- In UserManager, all data is accessed with synchronization
  - But still possible to see a user as neither active nor inactive
    - Therefore not thread-safe—can violate its specification!
  - Need to make compound operations atomic with respect to one other
    - Solution: synchronize UserManager methods

public class UserManager {
    // Each known user is in exactly one of {active, inactive}
    private final Set<User> active = Collections.synchronizedSet(...);
    private final Set<User> inactive = Collections.synchronizedSet(...);

    public synchronized void activate(User u) {
        if (inactive.remove(u))
            active.add(u);
    }
    public synchronized boolean isKnownUser(User u) {
        return active.contains(u) || inactive.contains(u);
    }
    public Set<User> getActiveUsers() {
        return Collections.unmodifiableSet(active);
    }
}
Encapsulate Data and Synchronization

- The problem was that synchronization was specified at a different level than the invariants
  - Result: atomicity failures (race conditions)
  - Could fix with client-side locking, but is fragile
  - Instead, encapsulate enforcement of invariants
    - All variables in an invariant should be guarded by same lock
    - Hold lock for duration of operation on related variables

- Always provide synchronization at the same level as the invariants
  - When composing operations on thread-safe objects, you may end up with multiple layers of synchronization
  - And that’s OK!
Summary: Encapsulation

- A thread-safe class encapsulates its data and any needed synchronization
  - Lack of encapsulation → fragility
- Without encapsulation, correctness and thread-safety can only describe the entire program, not a single class
- Wherever a class defines invariants on its state, it must provide synchronization to preserve those invariants
  - Even if this means multiple layers of synchronization
- Where should the synchronization go?
  - In the client—too fragile
  - In the component classes—may not preserve invariants
  - In the composite that defines invariants—just right
Agenda

Introduction

Rules for Writing Thread-Safe Code
- Document Thread-Safety Intent and Implementation
- Encapsulate Data and Synchronization
- **prefer Immutable Objects**
- Exploit Effective Immutability

Rules for Structuring Concurrent Applications
- Think Tasks, Not Threads
- Build Resource-Management Into Your Architecture
- Decouple Identification of Work from Execution

Rules for Improving Scalability
- Find and Eliminate the Serialization
Prefer Immutable Objects

- An immutable object is one whose
  - State cannot be changed after construction
  - All fields are final
    - Not optional—critical for thread-safety of immutable objects

- **Immutable objects are automatically thread-safe!**

- **Simpler**
  - Can only ever be in one state, controlled by the constructor

- **Safer**
  - Can be freely shared with unknown or malicious code, who cannot subvert their invariants

- **More scalable**
  - No synchronization required when sharing!

- (See Effective Java technology Item #13 for more)
Prefer Immutable Objects

- Most concurrency hazards stem from the need to coordinate access to mutable state
  - Race conditions and data races come from insufficient synchronization
  - Many other problems (e.g., deadlock) are consequences of strategies for proper coordination
- No mutable state → no need for coordination
  - No race conditions, data races, deadlocks, scalability bottlenecks
- Identify immutable objects with @Immutable
  - @Immutable implies @ThreadSafe
- Don’t worry about the cost of object creation
  - Object lifecycle is generally cheap
  - Immutable objects have some performance benefits too
Prefer Immutable Objects

- Even if immutability is not an option, less mutable state can still mean less coordination

- Benefits of immutability apply to individual variables as well as objects
  - Final fields have special visibility guarantees
  - Final fields are simpler than mutable fields

**Final is the new private**

- Declare fields final wherever practical
  - Worth doing extra work to avoid making fields nonfinal
  - In synchronization policy diagrams, final variables provide a synchronization policy for references
    - But not the referred-to object

- If you can’t get away with full immutability, seek to limit mutable state as much as possible
Agenda

Introduction

Rules for Writing Thread-Safe Code
- Document Thread-Safety Intent and Implementation
- Encapsulate Data and Synchronization
- Prefer Immutable Objects
- Exploit Effective Immutability

Rules for Structuring Concurrent Applications
- Think Tasks, Not Threads
- Build Resource-Management Into Your Architecture
- Decouple Identification of Work from Execution

Rules for Improving Scalability
- Find and Eliminate the Serialization
Find the Serialization

- Performance is a measure of *how fast*
  - Learning to work faster increases your performance

- Scalability is a measure of *how much more* work could be done with more resources
  - Learning to delegate increases your scalability

- When problems get over a certain size, performance improvements won’t get you there—you need to scale

- If a problem got ten times bigger, how much more resources would I need to solve it?
  - If you can just buy ten times as many CPUs (or memory or disks), then we say the problem scales *linearly or perfectly*
Find the Serialization

- Processor speeds flattened out around 2003
  - Moore's law now gives us more cores, not faster ones
  - Increasing throughput means keeping more cores busy
- Can no longer just buy a faster box to get a speedup
  - Must write programs that take advantage of additional CPUs
  - Just adding more cores may not improve throughput
    - Tasks must be amenable to parallelization

Source: (Graphic © 2006 Herb Sutter)
Find the Serialization

- System throughput is governed by **Amdahl’s Law**
  - Divides work into **serial** and **parallel** portions
    - Serial work cannot be sped up by adding resources
    - Parallelizable work can be

- Most tasks have a mix of serial and parallel work
  - Harvesting crops can be sped up with more workers
    - But additional workers will not make them grow any faster

- Amdahl’s Law says: \[ \text{Speedup} \leq \frac{1}{F + \frac{1-F}{N}} \]
  - \( F \) is the fraction that must be executed serially
  - \( N \) is the number of available workers

- As \( N \to \infty \), speedup \( \to 1/F \)
  - With 50% serialization, can only speed up by a factor of two
    - No matter how many processors
Find the Serialization

- Every task has some sources of serialization
  - You just have to know where to look

- The primary source of serialization is the **exclusive lock**
  - The longer locks are held for, the worse it gets

- Even when tasks consist only of thread-local computation, there is still serialization inherent in task dispatching

```java
while (!shutdownRequested) {
    Task t = taskQueue.take();  // potential serialization
    Result r = t.doTask();
    resultSet.add(result);      // potential serialization
}
```

- Accessing the task queue and the results container invariably involves serialization
Find the Serialization

- To improve scalability, you have to find the serialization and break it up
- Can reduce lock-induced serialization in several ways
  - Hold locks for less time—“get in, get out”
    - Move thread-local computation out of synchronized blocks
      - But don’t make them so small as to split atomic operations
    - Replace synchronized counters with AtomicInteger
  - Use lock splitting or lock striping to reduce lock contention
    - Guards different state with different locks
    - Reduces likelihood of lock contention
    - Replace synchronized Map with ConcurrentHashMap
Find the Serialization

- Can eliminate locking entirely in some cases
  - Replace mutable objects with immutable ones
  - Replace shared objects with thread-local ones
  - Confine objects to a specific thread (as in Swing)
  - Consider `ThreadLocal` for heavyweight mutable objects that don't need to be shared (e.g., `GregorianCalendar`)

- Signs that a concurrent program is bound by locking and not by CPU resources
  - Total CPU utilization < 100%
  - High percentage of kernel CPU usage
For More Information

- Other sessions
  - TS-2220: Testing Concurrent Software
  - TS-2007: Improving Software Quality with Static Analysis
  - BOF-2864: Debugging Data Races

- Books
  - *Java Concurrency in Practice* (Goetz, et al)
    - See [http://www.jcip.net](http://www.jcip.net)
  - *Concurrent Programming in Java* (Lea)
  - *Effective Java* (Bloch)
Q&A

Effective Concurrency for the Java Platform

Brian Goetz, Sun Microsystems
Effective Concurrency for the Java™ Platform

Brian Goetz
Senior Staff Engineer
Sun Microsystems, Inc.
brian.goetz@sun.com

TS-2388