



Chapter 10: Concurrent and Distributed Programming

10.2 Thread Safety

线程安全

Wang Zhongjie
rainy@hit.edu.cn

May 21, 2019

Outline

- Thread safety
- **Strategy 1: Confinement**
- **Strategy 2: Immutability**
- **Strategy 3: Using Threadsafe Data Types**
- **Strategy 4: Locks and Synchronization**
- **How to Make a Safety Argument**
- **Summary**

本章关注复杂软件系统的构造。这里的“复杂”包括两方面：

- (1) 多线程程序
- (2) 分布式程序

本节关注第一个方面：
如何设计threadsafe的ADT

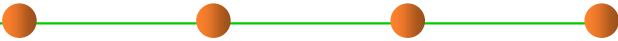
Reading

- MIT 6.031: 20
- CMU 17-214: Nov 8
- Java编程思想: 第21章
- Java Concurrency in Practice: 第1-5章
- Effective Java: 第10章
- 代码整洁之道: 第13章





1 What is Thread Safety



Thread Safety

- **Race conditions:** multiple threads sharing the same mutable variable without coordinating what they're doing.
- This is **unsafe**, because the correctness of the program may depend on accidents of timing of their low-level operations.
- 线程之间的“竞争条件”：作用于同一个mutable数据上的多个线程，彼此之间存在对该数据的访问竞争并导致interleaving，导致post-condition可能被违反，这是不安全的。

What **threadsafe** means

- A data type or static method is **threadsafe** if it behaves correctly when used from multiple threads, regardless of how those threads are executed, and without demanding additional coordination from the calling code. 线程安全: ADT或方法在多线程中要执行正确
- **How to catch the idea?**
 - “Behaves correctly” means satisfying its specification and preserving its rep invariant; 不违反spec、保持RI
 - “Regardless of how threads are executed” means threads might be on multiple processors or timesliced on the same processor; 与多少处理器、OS如何调度线程, 均无关
 - “Without additional coordination” means that the data type can’t put preconditions on its caller related to timing, like “you can’t call `get()` while `set()` is in progress.” 不需要在spec中强制要求client满足某种“线程安全”的义务

What **threadsafe** means

- Remember Iterator ? It's **not threadsafe**.
- Iterator 's specification says that **you can't modify a collection at the same time as you're iterating over it.**
- That's **a timing-related precondition** put on the caller, and Iterator makes no guarantee to behave correctly if you violate it.

```
public static void dropCourse6(ArrayList<String> subjects) {  
    MyIterator iter = new MyIterator(subjects);  
    while (iter.hasNext()) {  
        String subject = iter.next();  
        if (subject.startsWith("6.")) {  
            subjects.remove(subject);  
        }  
    }  
}
```

See Chapter 3.1

What **threadsafe** means: spec of **remove()**

- As a symptom of this non-local contract phenomenon, consider the Java collections classes, which are normally documented with very clear contracts on the client and implementer of a class.
 - Try to find where it documents the crucial requirement on the client that you can't modify a collection while you're iterating over it.

remove

```
void remove()
```

Removes from the underlying collection the last element returned by this iterator (optional operation). This method can be called only once per call to `next()`. The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.

Throws:

`UnsupportedOperationException` - if the `remove` operation is not supported by this iterator

`IllegalStateException` - if the `next` method has not yet been called, or the `remove` method has already been called after the last call to the `next` method

Four ways of threadsafe

Don't share: isolate mutable state in individual threads

Don't mutate: share only immutable state

If must share mutable state, use **threadsafe data type** or **synchronize**

- **Confinement** 限制数据共享. Don't share the variable between threads.
- **Immutability** 共享不可变数据. Make the shared data immutable.
- **Threadsafe data type** 共享线程安全的可变数据. Encapsulate the shared data in an existing threadsafe data type that does the coordination for you.
- **Synchronization** 同步机制: 通过锁的机制共享线程不安全的可变数据, 变并行为串行. Use synchronization to keep the threads from accessing the variable at the same time. Synchronization is what you need to build your own threadsafe data type.



Strategy 1: Confinement



Strategy 1: Confinement

- **Thread confinement is a simple idea:**

- You avoid races on mutable data by keeping that data confined to a single thread. 将可变数据限制在单一线程内部，避免竞争
- Don't give any other threads the ability to read or write the data directly. 不允许任何线程直接读写该数据

- **Since shared mutable data is the root cause of a race condition, confinement solves it by *not sharing* the mutable data. 核心思想：线程之间不共享mutable数据类型**

See section 8-1

- **Local variables are always thread confined.** A local variable is stored in the stack, and each thread has its own **stack**. There may be multiple invocations of a method running at a time, but each of those invocations has its own private copy of the variable, so the variable itself is confined.
- If a local variable is an **object reference**, you 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 (**not aliased**) from any other threads.

Strategy 1: Confinement

```
public class Factorial {

    /**
     * Computes n! and prints it on standard output.
     * @param n must be >= 0
     */
    private static void computeFact(final int n) {
        BigInteger result = new BigInteger("1");
        for (int i = 1; i <= n; ++i) {
            System.out.println("working on fact " + n);
            result = result.multiply(new BigInteger(String.valueOf(i)));
        }
        System.out.println("fact(" + n + ") = " + result);
    }

    public static void main(String[] args) {
        new Thread(new Runnable() { // create a thread using an
            public void run() {      // anonymous Runnable
                computeFact(99);
            }
        }).start();
        computeFact(100);
    }
}
```

Avoid Global Variables

- This class has a race in the `getInstance()` method – two threads could call it at the same time and end up creating **two copies** of the `PinballSimulator` object, which violates the rep invariant.

```
// This class has a race condition in it.
public class PinballSimulator {

    private static PinballSimulator simulator = null;
    // invariant: there should never be more than one PinballSimulator
    //             object created

    private PinballSimulator() {
        System.out.println("created a PinballSimulator object");
    }

    // factory method that returns the sole PinballSimulator object,
    // creating it if it doesn't exist
    public static PinballSimulator getInstance() {
        if (simulator == null) {
            simulator = new PinballSimulator();
        }
        return simulator;
    }
}
```

To fix this race using the thread confinement approach, you would specify that only a certain thread is allowed to call `getInstance()`.

But Java won't help you guarantee this.

Race
Condition

Singleton Design
Pattern
(see section 8-3)

Risk!

Avoid Global Variables

- Suppose two threads are running `getInstance()`.
- For each pair of possible line numbers that two threads are executing correspondingly, is it possible the invariant will be violated?

```
public class PinballSimulator {  
  
    private static PinballSimulator simulator = null;  
    // invariant: only one simulator object should be created  
  
    public static PinballSimulator getInstance() {  
1)        if (simulator == null) {  
2)            simulator = new PinballSimulator();  
        }  
3)        return simulator;  
    }  
  
    ...  
}
```

Lines 1 and 1

Lines 1 and 2

Lines 1 and 3

Java doesn't guarantee that the assignment to `simulator` in one thread will be immediately visible in other threads; it might be cached temporarily.

Avoid Global Variables

- **Global static variables are not automatically thread confined.**
 - If you have static variables in your program, then you have to make an argument that only one thread will ever use them, and you have to document that fact clearly. [\[Documenting in code – Chapter 4\]](#)
- **Better, you should eliminate the static variables entirely.**

Is this method threadsafe?

```
/**
 * @param x integer to test for primeness; requires x > 1
 * @return true if x is prime with high probability
 */
public static boolean isPrime(int x) {
    if (cache.containsKey(x)) return cache.get(x);
    boolean answer = BigInteger.valueOf(x).isProbablePrime(100);
    cache.put(x, answer);
    return answer;
}

private static Map<Integer, Boolean> cache = new HashMap<>();
```


Avoid Global Variables

- **`isPrime()`** method is not safe to call from multiple threads, and its clients may not even realize it.
 - The reason is that the `HashMap` referenced by the static variable `cache` is shared by all calls to `isPrime()`, and `HashMap` is not threadsafe.
 - If multiple threads mutate the map at the same time, by calling `cache.put()`, then the map can become corrupted in the same way that the bank account became corrupted.
 - If you're lucky, the corruption may cause an exception deep in the hash map, like a `NullPointerException` or `IndexOutOfBoundsException`.
 - But it also may just quietly give wrong answers.

```
public static boolean isPrime(int x) {  
    if (cache.containsKey(x)) return cache.get(x);  
    boolean answer = BigInteger.valueOf(x).isProbablePrime(100);  
    cache.put(x, answer);  
    return answer;  
}  
  
private static Map<Integer, Boolean> cache = new HashMap<>();
```

Example

In the following code, which variables are confined to a single thread?

```
public class C {  
    public static void main(...) {  
        new Thread(new Runnable() {  
            public void run() {  
                threadA();  
            }  
        }).start();  
  
        new Thread(new Runnable() {  
            public void run() {  
                threadB();  
            }  
        }).start();  
    }  
    private static String name  
        = "Napoleon Dynamite";  
    private static int cashLeft = 150;  
  
    private static void threadA() {  
        int amountA = 20;  
        cashLeft = spend(amountA);  
    }  
  
    private static void threadB() {  
        int amountB = 30;  
        cashLeft = spend(amountB);  
    }  
  
    private int spend (int amount) {  
        return cashLeft - amount;  
    }  
}
```

Confinement and Fields

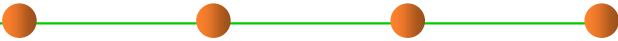
- If we want to argue that the `PinballSimulator` ADT is threadsafe, we cannot use confinement.
 - We don't know whether or how clients have created aliases to a `PinballSimulator` instance from multiple threads.
 - If they have, then concurrent calls to methods of this class will make concurrent access to its fields and their values.

```
public class PinballSimulator {  
  
    private final List<Ball> ballsInPlay;  
    private final Flipper leftFlipper;  
    private final Flipper rightFlipper;  
    // ... other instance fields...  
  
    // ... instance observer and mutator methods ...  
}
```

如果一个ADT的rep中包含mutable的属性且多线程之间对其进行mutator操作，那么就很难使用confinement策略来确保该ADT是线程安全的



Strategy 2: Immutability



Strategy 2: Immutability

- The second way of achieving thread safety is **by using immutable references and data types**. 使用不可变数据类型和不可变引用，避免多线程之间的 **race condition**
 - Immutability tackles the shared-mutable-data cause of a race condition and solves it simply by making the shared data *not mutable*.
- A variable declared **final** is unassignable and immutable references, so a variable declared **final** is safe to access from multiple threads.
 - You can only read the variable, not write it.
 - Because this safety applies only to the variable itself, and we still have to argue that the object the variable points to is immutable.

Strategy 2: Immutability

- Immutable objects are usually also threadsafe. 不可变数据通常是线程安全的
- We say “usually” because current definition of immutability is too loose for concurrent programming.
 - A type is immutable if an object of the type always represents the same abstract value for its entire lifetime.
 - But that actually allows the type the freedom to mutate its rep, as long as those mutations are invisible to clients, such as **beneficent mutation** (see Chapter 3.3).
 - Such as **caching, lazy computation, and data structure rebalancing**
- For concurrency, this kind of hidden mutation is not safe.
 - An immutable data type that uses beneficent mutation will have to make itself threadsafe using **locks**. 如果ADT中使用了**beneficent mutation**，必须要通过“加锁”机制来保证线程安全

Stronger definition of immutability

- In order to be confident that an immutable data type is threadsafe without locks, we need a stronger definition of immutability:
 - No **mutator** methods
 - All fields are **private** and **final**
 - No **representation exposure**
 - No **mutation** whatsoever of mutable objects in the rep – not even **beneficent mutation**
- If you follow these rules, then you can be confident that your immutable type will also be threadsafe.

Stronger definition of immutability

- Don't provide "setter" methods — methods that modify fields or objects referred to by fields.
- Make all fields **final** and **private**.
- Don't allow subclasses to **override** methods.
 - The simplest way to do this is to declare the class as **final**.
 - A more sophisticated approach is to make the constructor **private** and construct instances in **factory methods**.

Stronger definition of immutability

- **If the instance fields include references to mutable objects, don't allow those objects to be changed:**
 - Don't provide methods that modify the mutable objects.
 - Don't share references to the mutable objects.
 - Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies.
 - Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

Defensive copying (See Chapter 3.1)

Immutability and thread-safe

- Suppose you're reviewing an ADT which is specified to be immutable, to decide whether its implementation actually is immutable and threadsafe.
- Which of the following elements would you have to look at?

- Fields → Private and final?
- Creator implementations → Rep exposure (a reference to a mutable object passed in and stored in rep)?
- Producer implementations → Rep exposure (returning a reference to a mutable object in the rep)?
- Observer implementations → No mutators!
- Mutator implementations → Beneficent mutation? (for all operations)
- Client calls to creators → ADT must guarantee its own immutability, regardless of what clients do.
- Client calls to producers
- Client calls to observers
- Client calls to mutators

How about it?

- 相比起策略1 **Confinement**, 该策略2 **Immutability**允许有全局rep, 但是只能是**immutable**的。
- 如果一定需要**mutable**的ADT, 怎么办?



Strategy 3: Using Threadsafe Data Types



Strategy 3: Using Threadsafe Data Types

- The third major strategy for achieving thread safety is **to store shared mutable data in existing threadsafe data types**. 如果必须要用**mutable**的数据类型在多线程之间共享数据，要使用线程安全的数据类型。
 - When a data type in the Java library is threadsafe, its documentation will explicitly state that fact. 在JDK中的类，文档中明确指明了是否**threadsafe**
- It's become common in the Java API to find two mutable data types that do the same thing, one threadsafe and the other not. 一般来说，JDK同时提供两个相同功能的类，一个是**threadsafe**，另一个不是。原因：**threadsafe**的类一般性能上受影响
 - The reason is what this quote indicates: threadsafe data types usually incur a performance penalty compared to an unsafe type.

An example: StringBuffer vs. StringBuilder

Class StringBuffer

```
java.lang.Object
  java.lang.StringBuffer
```

All Implemented Interfaces:

Serializable, Appendable, CharSequence

```
public final class StringBuffer
extends Object
implements Serializable, CharSequence
```

A thread-safe, mutable sequence of characters. A `String`, but can be modified. At any point in time it contains a particular sequence of characters, but the length and the sequence can be changed through certain methods.

String buffers are safe for use by multiple threads. They are synchronized where necessary so that all the operations on a particular instance behave as if they occur in some sequential order consistent with the order of the method calls made by the individual threads involved.

Class StringBuilder

```
java.lang.Object
  java.lang.StringBuilder
```

All Implemented Interfaces:

Serializable, Appendable, CharSequence

```
public final class StringBuilder
extends Object
implements Serializable, CharSequence
```

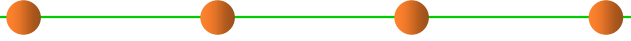
A mutable sequence of characters. This class provides an API compatible with `StringBuffer`, but with no guarantee of synchronization. This class is designed for use as a drop-in replacement for `StringBuffer` in places where the string buffer was being used by a single thread (as is generally the case). Where possible, it is recommended that this class be used in preference to `StringBuffer` as it will be faster under most implementations.

Threadsafe Collections

- The collection interfaces in Java – **List**, **Set**, **Map** – have basic implementations that are not threadsafe. 集合类都是线程不安全的
 - The implementations namely **ArrayList**, **HashMap**, and **HashSet**, cannot be used safely from more than one thread.
- The Collections API provides a set of wrapper methods to make collections threadsafe, while still mutable. Java API提供了进一步的 decorator
 - These wrappers effectively make each method of the collection atomic with respect to the other methods.
 - An atomic action effectively happens all at once – it doesn't interleave its internal operations with those of other actions, and none of the effects of the action are visible to other threads until the entire action is complete, so it never looks partially done.

```
private static Map<Integer, Boolean> cache =  
    Collections.synchronizedMap(new HashMap<>());
```

Threadsafe wrappers



```
public static <T> Collection<T>  
    synchronizedCollection(Collection<T> c);  
  
public static <T> Set<T> synchronizedSet(Set<T> s);  
  
public static <T> List<T> synchronizedList(List<T> list);  
  
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);  
  
public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s);  
  
public static <K,V> SortedMap<K,V>  
    synchronizedSortedMap(SortedMap<K,V> m);
```

Threadsafe wrappers

- Wrapper implementations delegate all their real work to a specified collection but add extra functionality on top of what this collection offers.
- This is an example of the **decorator pattern** (see section 5-3)
- These implementations are anonymous; rather than providing a public class, the library provides a static factory method.
- All these implementations are found in the Collections class, which consists solely of static methods.
- The synchronization wrappers add automatic synchronization (thread-safety) to an arbitrary collection.

An example

```
/**
 * @param x integer to test for primeness; requires x > 1
 * @return true if x is prime with high probability
 */
public static boolean isPrime(int x) {
    if (cache.containsKey(x)) return cache.get(x);
    boolean answer = BigInteger.valueOf(x).isProbablePrime(100);
    cache.put(x, answer);
    return answer;
}

private static Map<Integer, Boolean> cache = new HashMap<>();
```

- The global variable **cache** cause race condition among threads.

```
private static Map<Integer, Boolean> cache =
    Collections.synchronizedMap(new HashMap<>());
```

Don't circumvent the wrapper

- Make sure to throw away references to the underlying non-threadsafe collection, and access it only through the synchronized wrapper.
- The new `HashMap` is passed only to `synchronizedMap()` and never stored anywhere else.
- The underlying collection is still mutable, and code with a reference to it can circumvent immutability.
- 在使用`synchronizedMap(hashMap)`之后，不要再把参数`hashMap`共享给其他线程，不要保留别名，一定要彻底销毁

```
private static Map<Integer, Boolean> cache =  
    Collections.synchronizedMap(new HashMap<>());
```

Iterators are still not threadsafe

- Even though method calls on the collection itself (`get()`, `put()`, `add()`, etc.) are now threadsafe, iterators created from the collection are still not threadsafe. 即使在线程安全的集合类上，使用iterator也是不安全的
- So you can't use `iterator()`, or the for loop syntax:

```
for (String s: lst) { ... }
```

// not threadsafe, even if lst is a synchronized list wrapper
- The solution to this iteration problem will be to acquire the collection's **lock** when you need to iterate over it. 除非使用lock机制

```
List<Type> c = Collections.synchronizedList(new ArrayList<Type>());
```

```
synchronized(c) {           // to be introduced later (the 4-th threadsafe way)
    for (Type e : c)
        foo(e);
}
```

Atomic operations aren't enough to prevent races

- The way that you use the synchronized collection can still have a race condition. 即使是线程安全的collection类，仍可能产生竞争
 - 执行其上某个操作是threadsafe的，但如果多个操作放在一起，仍旧不安全

- Consider this code, which checks whether a list has at least one element and then gets that element:

```
if ( ! lst.isEmpty()) {  
    String s = lst.get(0);  
    ...  
}
```

两行语句可能产生interleaving

- Even if you make lst into a synchronized list, this code still may have a race condition, because another thread may remove the element between the isEmpty() call and the get() call.

Atomic operations aren't enough to prevent races

`Collections.synchronizedMap`



```
if (cache.containsKey(x))  
    return cache.get(x);
```

```
boolean answer = BigInteger.valueOf(x).isProbablePrime(100);  
cache.put(x, answer);
```

这些语句之间的
interleaving有无可能
产生race condition?

- The `synchronized` map ensures that `containsKey()`, `get()`, and `put()` are now **atomic**, so using them from multiple threads **won't damage the rep invariant** of the map.
- But those three operations can now interleave in arbitrary ways with each other, which **might break the invariant** that `isPrime` needs from the cache: If the cache maps an integer `x` to a value `f`, then `x` is prime if and only if `f` is true.
- If the cache ever fails this invariant, then we might return the wrong result.

A few points

- We have to argue that the races between `containsKey()` , `get()` , and `put()` don't threaten this invariant.
 - The race between `containsKey()` and `get()` is not harmful because we never remove items from the cache – once it contains a result for `x`, it will continue to do so.
 - There's a race between `containsKey()` and `put()`. As a result, it may end up that two threads will both test the primeness of the same `x` at the same time, and both will race to call `put()` with the answer. But both of them should call `put()` with the same answer, so it doesn't matter which one wins the race – the result will be the same.
- ...在注释中自证**threadsafe**
- The need to **make these kinds of careful arguments about safety** – even when you're using **threadsafe** data types – is the main reason that concurrency is hard.

Problem

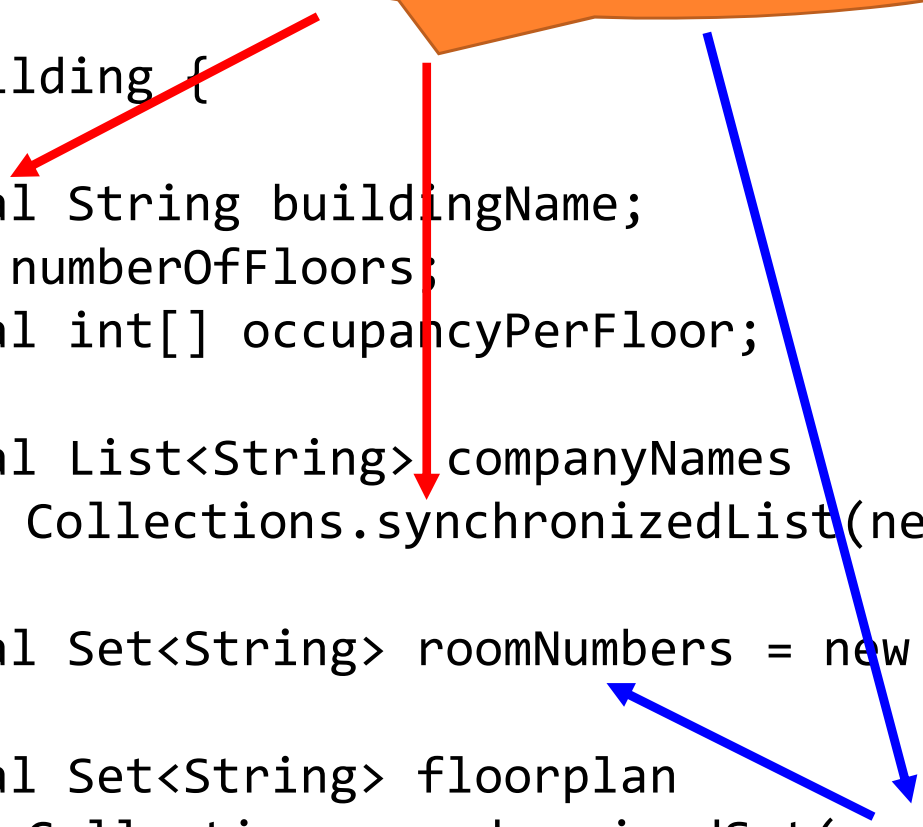
Which of these variables refer to a value of a threadsafe data type?

```
public class Building {  
    private final String buildingName;  
    private int numberOfFloors;  
    private final int[] occupancyPerFloor;  
  
    private final List<String> companyNames  
        = Collections.synchronizedList(new ArrayList<>());  
  
    private final Set<String> roomNumbers = new HashSet<>();  
  
    private final Set<String> floorplan  
        = Collections.synchronizedSet(roomNumbers);  
    ...  
}
```

Problem

Which of these variables are safe for use by multiple threads?

```
public class Building {  
    private final String buildingName;  
    private int numberOfFloors;  
    private final int[] occupancyPerFloor;  
  
    private final List<String> companyNames  
        = Collections.synchronizedList(new ArrayList<>());  
  
    private final Set<String> roomNumbers = new HashSet<>();  
  
    private final Set<String> floorplan  
        = Collections.synchronizedSet(roomNumbers);  
    ...  
}
```



Problem

Which of these variables cannot be involved in any race condition?

```
public class Building {  
    private final String buildingName;  
    private int numberOfFloors;  
    private final int[] occupancyPerFloor;  
  
    private final List<String> companyNames  
        = Collections.synchronizedList(new ArrayList<>());  
  
    private final Set<String> roomNumbers = new HashSet<>();  
  
    private final Set<String> floorplan  
        = Collections.synchronizedSet(roomNumbers);  
    ...  
}
```

Atomic operations
aren't enough to
prevent races

A short summary

- **Three major ways to achieve safety from race conditions on shared mutable data:**
 - Confinement: not sharing the data.
 - Immutability: sharing, but keeping the data immutable.
 - Threadsafe data types: storing the shared mutable data in a single threadsafe datatype.

A short summary

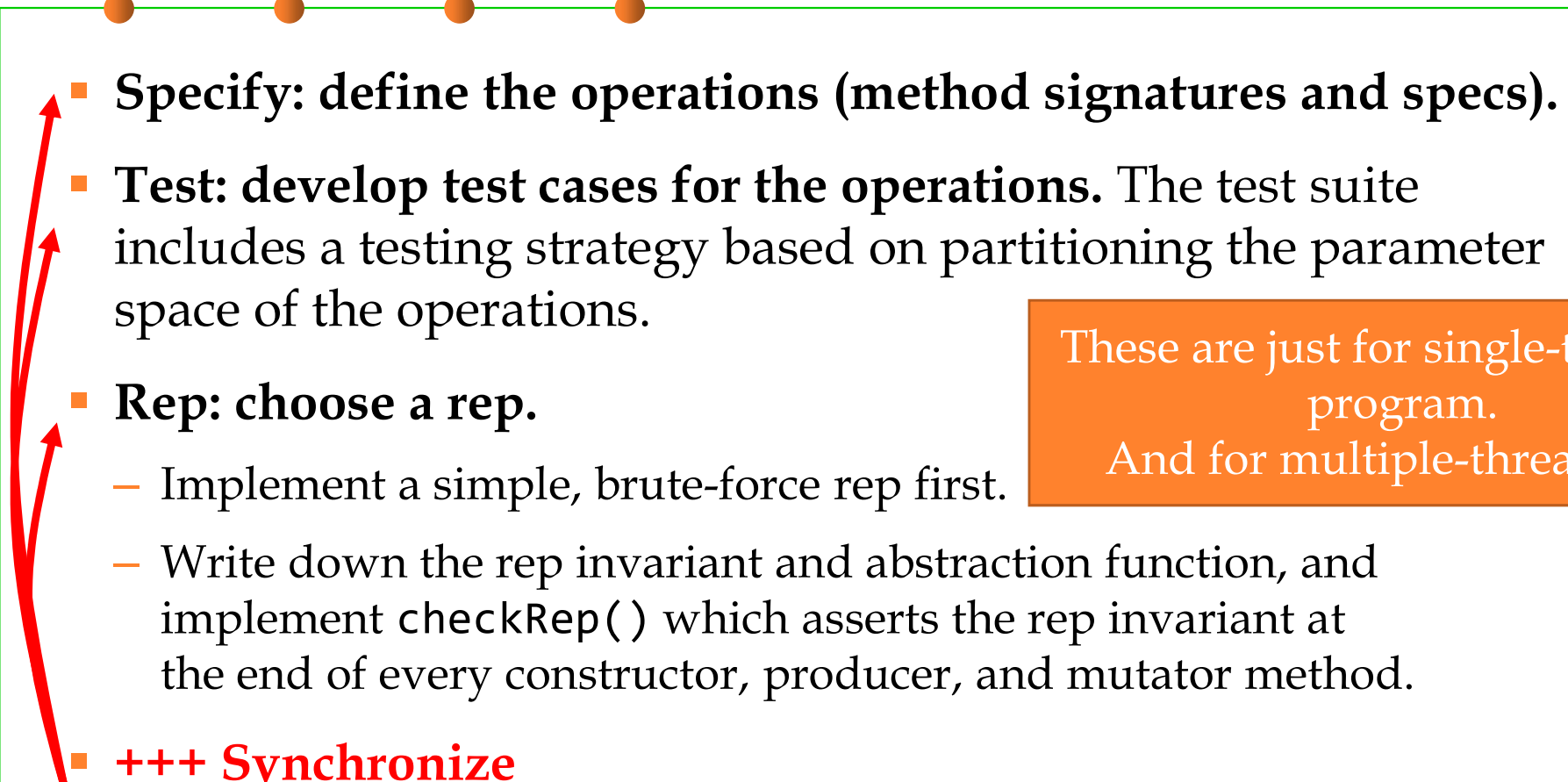
- **Safe from bugs.**
 - We're trying to eliminate a major class of concurrency bugs, race conditions, and eliminate them by design, not just by accident of timing.
- **Easy to understand.**
 - Applying these general, simple design patterns is far more understandable than a complex argument about which thread interleavings are possible and which are not.
- **Ready for change.**
 - We're writing down these justifications explicitly in a thread safety argument, so that maintenance programmers know what the code depends on for its thread safety.



5 How to Make a Safety Argument



Recall: Steps to develop an ADT

- 
- **Specify:** define the operations (method signatures and specs).
 - **Test:** develop test cases for the operations. The test suite includes a testing strategy based on partitioning the parameter space of the operations.
 - **Rep: choose a rep.**
 - Implement a simple, brute-force rep first.
 - Write down the rep invariant and abstraction function, and implement `checkRep()` which asserts the rep invariant at the end of every constructor, producer, and mutator method.
 - **+++ Synchronize**
 - Make an argument that your rep is threadsafe.
 - Write it down explicitly as a comment in your class, right by the rep invariant, so that a maintainer knows how you designed thread safety into the class.

These are just for single-threaded program.
And for multiple-threaded...

Make a safety argument

- **Concurrency is hard to test and debug !!!**
- So if you want to convince yourself and others that your concurrent program is correct, the best approach is to **make an explicit argument that it's free from races, and write it down.** 在代码中以注释的形式增加说明：该ADT采取了什么设计决策来保证线程安全
 - A safety argument needs to catalog all the threads that exist in your module or program, and the data that they use, and argue which of the four techniques you are using to protect against races for each data object or variable: **confinement, immutability, threadsafe data types, or synchronization.** 采取了四种方法中的哪一种？
 - When you use the last two, you also need to argue that **all accesses to the data are appropriately atomic – that is, that the invariants you depend on are not threatened by interleaving.** 如果是后两种，还需考虑对数据的访问都是原子的，不存在interleaving

Thread Safety Arguments for Confinement

- **Confinement is not usually an option** when we're making an argument just about a data type, because you have to know what threads exist in the system and what objects they've been given access to. 除非你知道线程访问的所有数据，否则**Confinement**无法彻底保证线程安全
 - If the data type creates its own set of threads, then you can talk about confinement with respect to those threads.
 - Otherwise, the threads are coming in from the outside, carrying client calls, and the data type may have no guarantees about which threads have references to what.
- **So confinement isn't a useful argument in that case.**
 - Usually we use confinement at a higher level, talking about the system as a whole and arguing why we don't need thread safety for some of our modules or data types, because they won't be shared across threads by design. 除非是在**ADT**内部创建的线程，可以清楚得知访问数据有哪些

Thread Safety Arguments for Immutability

```

/** MyString is an immutable data type representing a string of characters. */
public class MyString {
    private final char[] a;
    // Thread safety argument:
    //   This class is threadsafe because it's immutable:
    //   - a is final
    //   - a points to a mutable char array, but that array is encapsulated
    //     in this object, not shared with any other object or exposed to a
    //     client

```

We have to avoid rep exposure. Rep exposure is bad for any data type, since it threatens the data type's rep invariant.

It's also fatal to thread safety.

```

/** MyString is an immutable data type representing a string of characters. */
public class MyString {
    private final char[] a;
    private final int start;
    private final int len;
    // Rep invariant:
    //   0 <= start <= a.length
    //   0 <= len <= a.length-start
    // Abstraction function:
    //   represents the string of characters a[start],...,a[start+length-1]
    // Thread safety argument:
    //   This class is threadsafe because it's immutable:
    //   - a, start, and len are final
    //   - a points to a mutable char array, which may be shared with other
    //     MyString objects, but they never mutate it
    //   - the array is never exposed to a client

```

Is it really safe?

Bad Safety Arguments

```
/** MyStringBuffer is a threadsafe mutable string of characters. */
public class MyStringBuffer {
    private String text;
    // Rep invariant:
    //   none
    // Abstraction function:
    //   represents the sequence text[0],...,text[text.length()-1]
    // Thread safety argument:
    //   text is an immutable (and hence threadsafe) String,
    //   so this object is also threadsafe
}
```

- **Why doesn't this argument work?**
 - String is indeed immutable and threadsafe; but the rep pointing to that string, specifically the text variable, **is not immutable**.
 - text is not a **final** variable, and in fact it can't be **final** in this data type, because we need the data type to support insertion and deletion operations.
 - So reads and writes of the text variable itself are not threadsafe.
- **This argument is false.**

Bad Safety Arguments

```
public class Graph {  
    private final Set<Node> nodes =  
        Collections.synchronizedSet(new HashSet<>());  
    private final Map<Node, Set<Node>> edges =  
        Collections.synchronizedMap(new HashMap<>());  
  
    // Rep invariant:  
    //   for all x, y such that y is a member of edges.get(x),  
    //       x, y are both members of nodes  
    // Abstraction function:  
    //   represents a directed graph whose nodes are the set of nodes  
    //       and whose edges are the set (x,y) such that  
    //           y is a member of edges.get(x)  
    // Thread safety argument:  
    //   - nodes and edges are final, so those variables are immutable  
    //       and threadsafe  
    //   - nodes and edges point to threadsafe set and map data types
```

■ Is it a good safety argument?

- Graph relies on other threadsafe data types to help it implement its rep
- That prevents some race conditions, but not all, because the graph's rep invariant includes a relationship *between* the node set and the edge map.
All nodes that appear in the edge map also have to appear in the node set.

Bad Safety Arguments

```
public void addEdge(Node from, Node to) {  
    if ( ! edges.containsKey(from)) {  
        edges.put(from, Collections.synchronizedSet(new HashSet<>()));  
    }  
    edges.get(from).add(to);  
    nodes.add(from);  
    nodes.add(to);  
}
```

■ What if this code is executed?

- This code has a race condition in it. There is a crucial moment when the rep invariant is violated, right after the edges map is mutated, but just before the nodes set is mutated.
- Another operation on the graph might interleave at that moment, discover the rep invariant broken, and return wrong results.


Bad Safety Arguments

```
public void addEdge(Node from, Node to) {  
    if ( ! edges.containsKey(from)) {  
        edges.put(from, Collections.synchronizedSet(new HashSet<>()));  
    }  
    edges.get(from).add(to);  
    nodes.add(from);  
    nodes.add(to);  
}
```

- Even though the threadsafe set and map data types guarantee that their own `add()` and `put()` methods are atomic and noninterfering, they can't extend that guarantee to interactions between the two data structures. So the rep invariant of Graph is not safe from race conditions.
- Just using immutable and threadsafe-mutable data types is not sufficient when the rep invariant depends on relationships between objects in the rep.

Bad Safety Arguments

```
public void addEdge(Node from, Node to) {  
    if ( ! edges.containsKey(from)) {  
        edges.put(from, Collections.synchronizedSet(new HashSet<>()));  
    }  
    edges.get(from).add(to);  
    nodes.add(from);  
    nodes.add(to);  
}
```

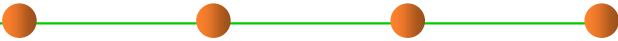


■ What if this code is executed?

- After the **if** check but before a new empty set has been stored in the **map**. That other thread calls **addEdge(from, to2)** to add an edge from the same **from** but going to a different **to**, so it adds its own empty set into the map and puts **to2** into it.
- On returning control back to the original thread, the original thread now overwrites that set with its own new empty set, discarding the edge from **from** to **to2**.



6 Summary



Summary

- This reading talked about three major ways to achieve safety from race conditions on shared mutable state:
 - Confinement: not sharing the variables or data.
 - Immutability: sharing, but keeping the data immutable and variables unreassignable.
 - Threadsafe data types: storing the shared mutable data in a single threadsafe datatype.

Summary

- These ideas connect to our three key properties of good software as follows:
 - Safe from bugs. We're trying to eliminate a major class of concurrency bugs, race conditions, and eliminate them by design, not just by accident of timing.
 - Easy to understand. Applying these general, simple design patterns is far more understandable than a complex argument about which thread interleavings are possible and which are not.
 - Ready for change. We're writing down these justifications explicitly in a thread safety argument, so that maintenance programmers know what the code depends on for its thread safety.



The end

May 21, 2019