

## Reading 20: Thread Safety

### What Threadsafe Means

#### Strategy 1: Confinement

#### Strategy 2: Immutability

#### Strategy 3: Using Threadsafe Data Types

#### How to Make a Safety Argument

#### Summary

# Reading 20: Thread Safety

## Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

## Objectives

Recall 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.

There are basically four ways to make variable access safe in shared-memory concurrency:

- **Confinement.** Don't share the variable between threads. This idea is called confinement, and we'll explore it today.
- **Immutability.** Make the shared data immutable. We've talked a lot about immutability already, but there are some additional constraints for concurrent programming that we'll talk about in this reading.
- **Threadsafe data type.** Encapsulate the shared data in an existing threadsafe data type that does the coordination for you. We'll talk about that today.
- **Synchronization.** 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.

We'll talk about the first three ways in this reading, along with how to make an argument that your code is threadsafe using those three ideas. We'll talk about the fourth approach, synchronization, in a later reading.

The material in this reading is inspired by an excellent book: Brian Goetz et al., *Java Concurrency in Practice* (<http://jcip.net/>), Addison-Wesley, 2006.

# What Threadsafe Means

## Reading 20: Thread Safety

### What Threadsafe Means

### Strategy 1: Confinement

### Strategy 2: Immutability

### Strategy 3: Using Threadsafe Data Types

### How to Make a Safety Argument

### Summary

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.

- “behaves correctly” means satisfying its specification and preserving its rep invariant;
- “regardless of how threads are executed” means threads might be on multiple processors or timesliced on the same processor;
- “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.”

Remember `Iterator` (<http://docs.oracle.com/javase/8/docs/api/?java/util/Iterator.html>)? 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.

## Strategy 1: Confinement

Our first way of achieving thread safety is *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.

Since shared mutable data is the root cause of a race condition, confinement solves it by *not sharing* the mutable data.

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 (in different threads or even at different levels of a single thread’s stack, if the method is recursive), but each of those invocations has its own private copy of the variable, so the variable itself is confined.

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.

Confinement is what makes the accesses to `n`, `i`, and `result` safe in code like this:

## Reading 20: Thread Safety

### What Threadsafe Means

#### Strategy 1: Confinement

#### Strategy 2: Immutability

#### Strategy 3: Using Threadsafe Data Types

#### How to Make a Safety Argument

#### Summary

```
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);
    }
}
```

Starting a thread with an **anonymous**

(<https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html>) Runnable is a common idiom. Read:

- A quick explanation of **using an anonymous Runnable to start a thread (anonymous-runnable/)**.

Let's look at snapshot diagrams for this code. Hover or tap on each step to update the diagram:

## Reading 20: Thread Safety

### What Threadsafe Means

#### Strategy 1: Confinement

#### Strategy 2: Immutability

#### Strategy 3: Using Threadsafe Data Types

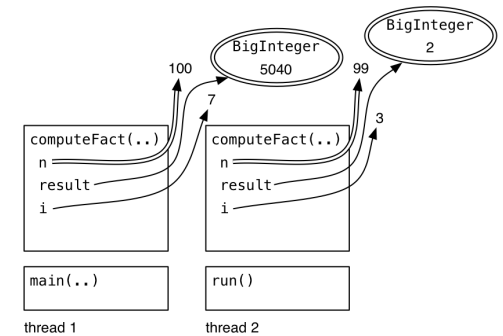
#### How to Make a Safety Argument

#### Summary

1. When we start the program, we start with one thread running `main`.
2. `main` creates a second thread using the anonymous `Runnable` idiom, and starts that thread.
3. At this point, we have two concurrent threads of execution. Their interleaving is unknown! But one *possibility* for the next thing that happens is that thread 1 enters `computeFact`.
4. Then, the next thing that *might* happen is that thread 2 also enters `computeFact`.

At this point, we see how **confinement** helps with thread safety: each execution of `computeFact` has its own `n`, `i`, and `result` variables. None of the objects they point to are mutable; if they were mutable, we would need to check that the objects are not aliased from other threads.

5. The `computeFact` computations proceed independently, updating their respective variables



## Avoid Global Variables

Unlike local variables, 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. Better, you should eliminate the static variables entirely.

Here's an example:

## Reading 20: Thread Safety

### What Threadsafe Means

#### Strategy 1: Confinement

#### Strategy 2: Immutability

#### Strategy 3: Using Threadsafe Data Types

#### How to Make a Safety Argument

#### Summary

```
// 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;
    }
}
```

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 we don't want.

To fix this race using the thread confinement approach, you would specify that only a certain thread (maybe the “pinball simulation thread”) is allowed to call `PinballSimulator.getInstance()`. The risk here is that Java won't help you guarantee this.

In general, static variables are very risky for concurrency. They might be hiding behind an innocuous function that seems to have no side-effects or mutations. Consider this example:

## Reading 20: Thread Safety

### What Threadsafe Means

#### Strategy 1: Confinement

#### Strategy 2: Immutability

#### Strategy 3: Using Threadsafe Data Types

#### How to Make a Safety Argument

#### Summary

```
// is this method threadsafe?
/**
 * @param x integer to test for primeness; requires x > 1
 * @return true if and only if x is prime
 */
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<>();
```

This function stores the answers from previous calls in case they're requested again. This technique is called memoization (<http://en.wikipedia.org/wiki/Memoization>), and it's a sensible optimization for slow functions like exact primality testing. But now the `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 in the last reading ([http://web.mit.edu/6.005/www/fa15/classes/19-concurrency/#shared\\_memory\\_example](http://web.mit.edu/6.005/www/fa15/classes/19-concurrency/#shared_memory_example)). 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, as we saw in the bank account example ([http://web.mit.edu/6.005/www/fa15/classes/19-concurrency/#shared\\_memory\\_example](http://web.mit.edu/6.005/www/fa15/classes/19-concurrency/#shared_memory_example)).

## READING EXERCISES

### Factorial

In the factorial example above, `main` looks like:

## Reading 20: Thread Safety

### What Threadsafe Means

#### Strategy 1: Confinement

#### Strategy 2: Immutability

#### Strategy 3: Using Threadsafe Data Types

#### How to Make a Safety Argument

#### Summary

```
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);  
}
```

Which of the following are possible interleavings?

- ✘ ☒ The call to `computeFact(100)` starts before the call to `computeFact(99)` starts ☒
- ☐ The call to `computeFact(99)` starts before the call to `computeFact(100)` starts ☒
- ☐ The call to `computeFact(100)` finishes before the call to `computeFact(99)` starts ☒
- ☐ The call to `computeFact(99)` finishes before the call to `computeFact(100)` starts ☒
- ☐ All of these are possible.

CHECK

EXPLAIN

## Pinball Simulator

Here's part of the pinball simulator example above:

## Reading 20: Thread Safety

### What Threadsafe Means

#### Strategy 1: Confinement

#### Strategy 2: Immutability

#### Strategy 3: Using Threadsafe Data Types

#### How to Make a Safety Argument

#### Summary

```
public class PinballSimulator {  
  
    private static PinballSimulator simulator = null;  
  
    // ...  
  
    public static PinballSimulator getInstance() {  
1)         if (simulator == null) {  
2)             simulator = new PinballSimulator();  
            }  
3)         return simulator;  
    }  
}
```

The code has a race condition that invalidates the invariant that only one simulator object is created.

Suppose two threads are running `getInstance()`. One thread is about to execute one of the numbered lines above; the other thread is about to execute the other. For each pair of possible line numbers, is it possible the invariant will be violated?

#### About to execute lines 1 and 3

- ✓ ☒ Yes, it could be violated ✓  
☐ No, we're safe

- The thread on line 3 has already assigned simulator, so the thread on line 1 will not enter the conditional. Right?

**Unfortunately, that's not correct.** As we saw in the last reading (<http://web.mit.edu/6.005/www/fa15/classes/19-concurrency/#reordering>), Java doesn't guarantee that the assignment to `simulator` in one thread will be immediately visible in other threads; it might be cached temporarily. In fact, our reasoning is broken, and the invariant can still be violated.

#### About to execute lines 1 and 2

- ✗ ☐ Yes, it could be violated ✓  
☒ No, we're safe



## Reading 20: Thread Safety

### What Threadsafe Means

#### Strategy 1: Confinement

#### Strategy 2: Immutability


#### Strategy 3: Using Threadsafe Data Types

#### How to Make a Safety Argument

#### Summary

- If the thread about to execute line 1 goes first, both threads are inside the conditional and will create new simulator objects.

#### About to execute lines 1 and 1

- ✕ ☐ Yes, it could be violated 
- ☒ No, we're safe

- If both threads test the predicate before either thread assigns `simulator`, both will enter the conditional and create new simulator objects.

CHECK

EXPLAIN

Confinement

## Strategy 2: Immutability

Our second way of achieving thread safety is 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.

Immutable objects are usually also threadsafe. We say “usually” here because our current definition of immutability is too loose for concurrent programming. We’ve said that 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. We saw an example of this notion, called benevolent or beneficent mutation, when we looked at an immutable list that cached its length in a mutable field ([http://web.mit.edu/6.005/www/fa15/classes/16-recursive-data-types/recursive/#tuning\\_the\\_rep](http://web.mit.edu/6.005/www/fa15/classes/16-recursive-data-types/recursive/#tuning_the_rep)) the first time the length was requested by a client. Caching is a typical kind of beneficent mutation.

## Reading 20: Thread Safety

### What Threadsafe Means

#### Strategy 1: Confinement

#### Strategy 2: Immutability

#### Strategy 3: Using Threadsafe Data Types

#### How to Make a Safety Argument

#### Summary

For concurrency, though, this kind of hidden mutation is not safe. An immutable data type that uses beneficent mutation will have to make itself threadsafe using locks (the same technique required of mutable data types), which we'll talk about in a future reading.

## Stronger definition of immutability

So 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 (<http://web.mit.edu/6.005/www/fa15/classes/13-abstraction-functions-rep-invariants/#invariants>)
- no mutation whatsoever of mutable objects in the rep – not even beneficent mutation ([http://web.mit.edu/6.005/www/fa15/classes/16-recursive-data-types/recursive/#tuning\\_the\\_rep](http://web.mit.edu/6.005/www/fa15/classes/16-recursive-data-types/recursive/#tuning_the_rep))

If you follow these rules, then you can be confident that your immutable type will also be threadsafe.

In the Java Tutorials, read:




- **A Strategy for Defining Immutable Objects** (<http://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html>) (1 page)

## READING EXERCISES

### Immutability

Suppose you're reviewing an abstract data type 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 
- ☐ creator implementations 
- ☒ client calls to creators
- ☐ producer implementations 

## Reading 20: Thread Safety

### What Threadsafe Means


#### Strategy 1: Confinement

#### Strategy 2: Immutability

#### Strategy 3: Using Threadsafe Data Types

#### How to Make a Safety Argument

#### Summary

- ☐ client calls to producers
- ☒ observer implementations 
- ☐ client calls to observers
- ☐ mutator implementations
- ☐ client calls to mutators



Fields need to be examined to make sure they're private and final.

Creator and producer implementations need to be checked for rep exposure – e.g., to make sure that a reference to a mutable object passed in from a client isn't being stored in the rep.

Observer implementations need to be checked for rep exposure as well, in this case returning a reference to a mutable object in the rep.

Client calls to operations do not need to be examined, because the ADT must guarantee its own immutability, regardless of what clients do.

Mutators do not need to be examined, because an ostensibly-immutable ADT shouldn't have any mutators.

CHECK

EXPLAIN

## Strategy 3: Using Threadsafe Data Types

Our third major strategy for achieving thread safety is to store shared mutable data in existing threadsafe data types.

When a data type in the Java library is threadsafe, its documentation will explicitly state that fact. For example, here's what `StringBuffer` (<http://docs.oracle.com/javase/8/docs/api/java/lang/StringBuffer.html>) says:

## Reading 20: Thread Safety

### What Threadsafe Means

#### Strategy 1: Confinement

#### Strategy 2: Immutability

#### Strategy 3: Using Threadsafe Data Types

#### How to Make a Safety Argument

#### Summary

[StringBuffer is] A thread-safe, mutable sequence of characters. A string buffer is like a String, but can be modified. At any point in time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.

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

This is in contrast to StringBuilder (<http://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html>):

[StringBuilder is] 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.

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. The reason is what this quote indicates: threadsafe data types usually incur a performance penalty compared to an unsafe type.

It's deeply unfortunate that `StringBuffer` and `StringBuilder` are named so similarly, without any indication in the name that thread safety is the crucial difference between them. It's also unfortunate that they don't share a common interface, so you can't simply swap in one implementation for the other for the times when you need thread safety. The Java collection interfaces do much better in this respect, as we'll see next.

## Threadsafe Collections

The collection interfaces in Java – `List`, `Set`, `Map` – have basic implementations that are not threadsafe. The implementations of these that you've been used to using, namely `ArrayList`, `HashMap`, and `HashSet`, cannot be used safely from more than one thread.

Fortunately, just like the Collections API provides wrapper methods that make collections immutable, it provides another set of wrapper methods to make collections threadsafe, while still mutable.

## Reading 20: Thread Safety

### What Threadsafe Means

#### Strategy 1: Confinement

#### Strategy 2: Immutability

#### Strategy 3: Using Threadsafe Data Types

#### How to Make a Safety Argument

#### Summary

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.

Now we see a way to fix the `isPrime()` method we had earlier in the reading:

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

A few points here.

**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. That happens automatically in the line of code above, since the new `HashMap` is passed only to `synchronizedMap()` and never stored anywhere else. (We saw this same warning with the unmodifiable wrappers: the underlying collection is still mutable, and code with a reference to it can circumvent immutability.)

**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. 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, which we'll talk about in a future reading.

Finally, **atomic operations aren't enough to prevent races**: the way that you use the synchronized collection can still have a race condition. 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); ... }
```

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.

Even the `isPrime()` method still has potential races:

## Reading 20: Thread Safety

### What Threadsafe Means

#### Strategy 1: Confinement

#### Strategy 2: Immutability

#### Strategy 3: Using Threadsafe Data Types

#### How to Make a Safety Argument

#### Summary

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

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.

So we have to argue that the races between `containsKey()`, `get()`, and `put()` don't threaten this invariant.

1. 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.
2. 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.

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.

In the Java Tutorials, read:

- **Wrapper Collections**  
(<http://docs.oracle.com/javase/tutorial/collections/implementations/wrapper.html>) (1 page)
- **Concurrent Collections**  
(<http://docs.oracle.com/javase/tutorial/essential/concurrency/collections.html>) (1 page)

## READING EXERCISES

Threadsafe data types

## Reading 20: Thread Safety

### What Threadsafe Means

#### Strategy 1: Confinement

#### Strategy 2: Immutability

#### Strategy 3: Using Threadsafe Data Types




#### How to Make a Safety Argument

#### Summary

Consider this class's rep:



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

**Which of these variables use a threadsafe data type?**

- ✘ ☐ buildingName 
- ☐ numberOfFloors 
- ☐ occupancyPerFloor
- ☐ companyNames 


- String and int are both immutable, so the types themselves are threadsafe.
- int[] is mutable, and not threadsafe.
- List<String> is not automatically threadsafe, but the implementation of List<String> used here is a synchronized list wrapper, and companyNames is final so it can never be assigned to a different List, so this type is threadsafe.

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

- ✘ ☐ buildingName 
- ☒ numberOfFloors
- ☐ occupancyPerFloor
- ☐ companyNames 

- Not only does the variable's type have to be threadsafe, but the variable itself should be final. buildingName and companyNames satisfy that, but reads and writes of numberOfFloors may have race conditions.

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

- ☐ buildingName 

## Reading 20: Thread Safety

### What Threadsafe Means

#### Strategy 1: Confinement

#### Strategy 2: Immutability

#### Strategy 3: Using Threadsafe Data Types

### How to Make a Safety Argument

#### Summary

- ✗ ☒ numberOfFloors
- ☐ occupancyPerFloor
- ☐ companyNames

- buildingName is immutable, so it can't be involved in any race condition.
- companyNames might still be involved in a race condition caused by (otherwise safe) mutations to the list, e.g.:

```
if (companyNames.size() > 0) { String firstCompany = companyNames.get(0); }
```

If another thread could empty the companyNames list between the size check and the get call, then this code will fail.

CHECK

EXPLAIN

## How to Make a Safety Argument

We've seen that 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.

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. 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. We gave one of those arguments for `isPrime` above.

## Thread Safety Arguments for Data Types

Let's see some examples of how to make thread safety arguments for a data type. Remember our four approaches to thread safety: confinement, immutability, threadsafe data types, and synchronization. Since we haven't talked about synchronization in this reading, we'll just focus on the first three approaches.



## Reading 20: Thread Safety

### What Threadsafe Means

#### Strategy 1: Confinement

#### Strategy 2: Immutability

#### Strategy 3: Using Threadsafe Data Types

#### How to Make a Safety Argument

#### Summary

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. 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.

Immutability is often a useful argument:

```
/** 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
}
```

Here's another rep for MyString that requires a little more care in the argument:

## Reading 20: Thread Safety

### What Threadsafe Means

#### Strategy 1: Confinement

#### Strategy 2: Immutability

#### Strategy 3: Using Threadsafe Data Types

#### How to Make a Safety Argument

#### Summary

```
/** 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
    //      String objects, but they never mutate it
    //    - the array is never exposed to a client
```

Note that since this `MyString` rep was designed for sharing the array between multiple `MyString` objects, we have to ensure that the sharing doesn't threaten its thread safety. As long as it doesn't threaten the `MyString`'s immutability, however, we can be confident that it won't threaten the thread safety.

We also 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.

## Bad Safety Arguments

Here are some *incorrect* arguments for thread safety:

## Reading 20: Thread Safety

### What Threadsafe Means

#### Strategy 1: Confinement

```
/** StringBuffer is a threadsafe mutable string of characters. */
public class StringBuffer {
    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
}
```

#### Strategy 2: Immutability

#### Strategy 3: Using Threadsafe Data Types

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.

Here's another broken argument:

#### How to Make a Safety Argument

#### Summary

```
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
}
```

This is a graph data type, which stores its nodes in a set and its edges in a map. (Quick quiz: is `Graph` a mutable or immutable data type? What do the final keywords have to do with its mutability?) `Graph` relies on other threadsafe data types to help it implement its rep – specifically the threadsafe `Set` and

## Reading 20: Thread Safety

### What Threadsafe Means

#### Strategy 1: Confinement

#### Strategy 2: Immutability

#### Strategy 3: Using Threadsafe Data Types

#### How to Make a Safety Argument

#### Summary

Map wrappers that we talked about above. 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. So there may be code like this:

```
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);
}
```

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. 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.

We'll have to fix this with synchronization, and we'll see how in a future reading.

## READING EXERCISES

### Safety arguments

Consider the following ADT with a **bad** safety argument that appeared above:

## Reading 20: Thread Safety

### What Threadsafe Means

#### Strategy 1: Confinement

#### Strategy 2: Immutability

#### Strategy 3: Using Threadsafe Data Types

#### How to Make a Safety Argument

#### Summary

```
/** StringBuffer is a threadsafe mutable string of characters. */
public class StringBuffer {
    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

    /** @return the string represented by this buffer,
     *      with all letters converted to uppercase */
    public String toUpperCase() { return text.toUpperCase(); }

    /** @param pos position to insert text into the buffer,
     *      requires 0 <= pos <= length of the current string
     *      @param s text to insert
     *      Mutates this buffer to insert s as a substring at position pos.
     */
    public void insert(int pos, String s) {
        text = text.substring(0, pos) + s + text.substring(pos);
    }

    /** @return the string represented by this buffer */
    public void toString() { return text; }

    /** Resets this buffer to the empty string. */
    public void clear() { text = ""; }

    /** @return the first character of this buffer, or "" if this buffer
     is empty */
    public String first() {
        if (text.length() > 0) {
            return String.valueOf(text.charAt(0));
        } else {
            return "";
        }
    }
}
```

## Reading 20: Thread Safety

### What Threadsafe Means

#### Strategy 1: Confinement

#### Strategy 2: Immutability

#### Strategy 3: Using Threadsafe Data Types

#### How to Make a Safety Argument

#### Summary

```
}  
}
```

Which of these methods are counterexamples to the buggy safety argument, because they have a race condition?

In particular, you should mark method `A` as a counterexample if it's possible that, if one thread is running method `A` at the same time as another thread is running some other method, some interleaving would violate `A`'s postcondition:

- ✘ ☐ `toUpperCase`  
☐ `insert` ✓  
☐ `toString`  
☒ `clear`  
☐ `first` ✓

➤ Both `insert` and `first` may fail if `clear` interleaves at the wrong moment – they will throw `IndexOutOfBoundsException` exceptions, which are not permitted by their specs, assuming their precondition was satisfied when the method started.

`toUpperCase`, `toString`, and `clear` will not violate their own postconditions.

`toUpperCase` and `toString` are just observers, in fact, so they won't be able to hurt other methods by interleaving with them.

CHECK

EXPLAIN

## Serializability

Look again at the code for the exercise above. We might also be concerned that `clear` and `insert` could interleave such that a client sees `clear` violate its postcondition.

Suppose two threads are sharing `StringBuffer sb` representing `"z"`. They

**A**

call `sb.clear()`

**B**

call `sb.insert(0, "a")`

## Reading 20: Thread Safety

### What Threadsafe Means

#### Strategy 1: Confinement

#### Strategy 2: Immutability

#### Strategy 3: Using Threadsafe Data Types

#### How to Make a Safety Argument

#### Summary

run `clear` and `insert` concurrently as shown on the right.

Thread A's assertion will fail, but not because `clear` violated its postcondition. Indeed, when all the code in `clear` has finished running, the postcondition is satisfied.

The real problem is that thread A has not anticipated possible interleaving between `clear()` and the `assert`. With any threadsafe mutable type where atomic mutators are called concurrently, *some* mutation has to “win” by being the last one applied. The result that thread A observed is identical to the execution below, where the mutators don't interleave at all:

A	B
call <code>sb.clear()</code>	
— in <code>clear</code> : <code>text = ""</code>	
— <code>clear</code> returns	
	call <code>sb.insert(0, "a")</code>
	— in <code>insert</code> : <code>text = "" + "a" + "z"</code>
	— <code>insert</code> returns
<code>assert sb.toString()</code> <code>.equals("")</code>	

What we demand from a threadsafe data type is that when clients call its atomic operations concurrently, the results are consistent with *some* sequential ordering of the calls. In this case, clearing and inserting, that means either `clear` -followed-by- `insert`, or `insert` -followed-by- `clear`. This

property is called **serializability** (<https://en.wikipedia.org/wiki/Serializability>): for any set of operations executed concurrently, the result (the values and state observable by clients) must be a result given by *some* sequential ordering of those operations.

## Reading 20: Thread Safety

### What Threadsafe Means

#### Strategy 1: Confinement

#### Strategy 2: Immutability

#### Strategy 3: Using Threadsafe Data Types

#### How to Make a Safety Argument

#### Summary

## READING EXERCISES

### Serializability

Suppose two threads are sharing a `StringBuffer` representing `"z"`.

For each pair of concurrent calls and their result, does that outcome violate serializability (and therefore demonstrate that `StringBuffer` is not threadsafe)?

`clear()` and `insert(0, "a")` → `insert` throws an `IndexOutOfBoundsException`

- ✗ ☐ Violates serializability ☒
- ☒ Consistent with serializability

➤ Calling `insert` with `pos=0` should never violate its precondition, so throwing the exception is not consistent with any sequential ordering.

`clear()` and `insert(1, "a")` → `insert` throws an `IndexOutOfBoundsException`

- ✓ ☐ Violates serializability ☒
- ☒ Consistent with serializability

➤ This is consistent with `clear`-followed-by-`insert`. The precondition `pos <= length` of the current string is violated, since `clear` happened first.

`first()` and `insert(0, "a")` → `first` returns `"a"`

- ✓ ☐ Violates serializability ☒
- ☒ Consistent with serializability

➤ This is consistent with `insert`-followed-by-`first`.

`first()` and `clear()` → `first` returns `"z"`

- ✗ ☒ Violates serializability

You are not logged in.



## Reading 20: Thread Safety

### What Threadsafe Means

#### Strategy 1: Confinement

#### Strategy 2: Immutability

#### Strategy 3: Using Threadsafe Data Types

#### How to Make a Safety Argument

##### Summary

Collaboratively authored with contributions from: Saman Amarasinghe, Adam Chlipala, Srini Devadas, Michael Ernst, Max Goldman, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama. This work is licensed under CC BY-SA 4.0 (<http://creativecommons.org/licenses/by-sa/4.0/>).

☐ Consistent with serializability ✓

➤ This is consistent with `first`-followed-by-`clear`.

`first()` and `clear()` → `first` throws an `IndexOutOfBoundsException`

✓ ☒ Violates serializability ✓

☐ Consistent with serializability

➤ The exception is never allowed by `first`'s postcondition, so throwing it is not consistent with any sequential ordering.

CHECK

EXPLAIN

## Summary

This reading talked about 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.

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.