Practical Concurrent and Parallel Programming III

Shared Memory II

Raúl Pardo

Assignment workload

- We would like to get an estimation on the amount of hours you spend on assignments
- Please go to the following mentimeter poll

  https://www.menti.com/ale2q4ottxor

Previously on PCPP…

- Readers and Writers Problem
- Monitors
- Fairness
- Java Intrinsic Locks (**`synchronized`**)
- Hardware and Programming Language Concurrency Issues
  - Visibility
  - Reordering
- Volatile variables (**`volatile`**)

Agenda

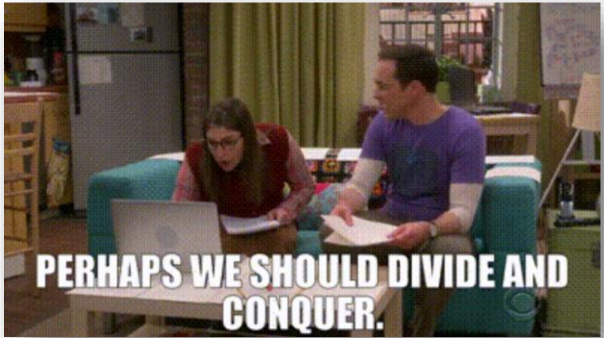
- Definitions of thread-safety
  - Classes
  - Programs
- Safe publication
- Immutability
- Instance confinement
- Synchronization primitives (synchronizers)
  - Semaphores
  - Barriers
- Producer-consumer problem

Thread safety

- We have already covered the basic concepts to analyse concurrent programs
- Analysing concurrent programs is tricky
  - You have experienced this already in the assignments where you work with programs consisting in a few lines of code
- Imagine having to reason about applications with hundreds of lines of code and many classes
  - Server applications
  - Operating Systems
  - GUIs
  - …

Thread safety

- We have already covered the basic concepts to analyse concurrent programs
- Analysing concurrent programs is tricky
  - You have experienced this already in the assignments where you work with programs consisting in a few lines of code
- Imagine having to reason about applications with hundreds of lines of code and many classes
  - Server applications
  - Operating Systems
  - GUIs
  - …



Thread-safe classes

- It is more manageable to separately analyse parts of the code and then combine them in safe ways
- In Object Oriented languages (such as Java) we can focus on analysing thread-safety for classes
- This reduces the analysis to concurrent method calls and field accesses

Thread-safe classes

*A **class** is said to be **thread-safe** if and only if no concurrent execution of method calls or field accesses (read/write) result in race conditions*

PCPP teaching team

Thread-safe classes

*A **class** is said to be **thread-safe** if and only if*
no concurrent execution of
method calls or field accesses (read/write)
result in race conditions

Note that this definition is independent of class invariants as opposed to Goetz Chapter 4. This definition is more similar to Goetz Chapter 2, page 18.

PCPP teaching team

Thread-safe classes

<u>WARNING</u>: Note that, in this course, *thread-safety* is not an umbrella term for code that seem to behave correctly in concurrent environments.

*A **<u>class</u>** is said to be **<u>thread-safe</u>** if and only if no concurrent execution of method calls or field accesses (read/write) result in race conditions*

Note that this definition is <u>independent of class invariants</u> as opposed to Goetz Chapter 4. This definition is more <u>similar to Goetz Chapter 2, page 18</u>.

PCPP teaching team

Thread-safe program

*A concurrent **program** is said to be **thread-safe***
if and only if it is race condition free

Do not confuse thread-safe classes with thread-safe programs. Thread-safe programs are not defined in Goetz. But it is aligned with the definition of [correctly synchronized programs in JLS]

PCPP teaching team

Thread-safety

It is very important to note that:

For any program p,

p only accesses thread-safe <u>classes</u>

$$\not\Rightarrow$$

p is a thread-safe <u>program</u>

Thread-safety

It is very important to note that:

For any program p,

p only accesses thread-safe <u>classes</u>

$$\not\Rightarrow$$

p is a thread-safe <u>program</u>

Programs using thread-safe classes may contain race conditions.

Thread-safety

Does this hold?

p is a thread-safe <u>program</u>

$\nRightarrow$

p only accesses thread-safe <u>classes</u>

It is very importa

For any progra

p only accesses thread-safe <u>classes</u>

$\nRightarrow$

p is a thread-safe <u>program</u>

Programs using thread-safe classes may contain race conditions.

Thread-safety

It is very

This implication is a bit informal/ambiguous. A way to read the implication is: “Can we have thread-safe programs if we are not using thread-safe classes?” And the answer is yes. For instance programs that do not work on shared memory, or that only read data (from non thread-safe classes). In other words, you may use a non-thread safe class in a thread safe manner. In these cases, you have a thread-safe program without using thread-safe classes.

thread-safe class $\not\Rightarrow$ *thread-safe program*

Programs using thread-safe classes may contain race conditions.

Thread-safe classes

- To analyse whether a class is thread-safe, we must identify/consider:
  - Class state
  - Escaping
  - (Safe) publication
  - Immutability
  - Mutual exclusion

Class state

- As we have seen, (uncontrolled) concurrent access to the shared state (variables) may lead to race conditions
- So, the first thing we need to do is to identify the fields that may be shared by several threads
- The state of a class involves the fields defined in the class
  - In a nutshell, our goal is to ensure that concurrent manipulation of the class state is race condition free

```
class C {
    // class state (variables)
    T s1;
    T s2;
    T s3;
    T s4;
    …

    // class methods
    T m1(…)  {…}
    T m2(…)  {…}
    T m3(…)  {…}
    …
}
```

Class state

If a class has no state (variables), is it thread-safe?

- As we have seen, (uncontrolled) concurrent access to the shared state (variables) may lead to race conditions

- So, the first thing we need to do is to identify the fields that may be shared by several threads

- The state of a class involves the fields defined in the class
    - In a nutshell, our goal is to ensure that concurrent manipulation of the class state is race condition free

```
class C {
     // class state (variables)
     T s1;
     T s2;
     T s3;
     T s4;
     …

     // class methods
     T m1(…)  {…}
     T m2(…)  {…}
     T m3(…)  {…}
     …
}
```

Class state

Assuming that all methods are *pure* (the output only depends on the input), then yes. This is because there is no shared state.

- As we have seen, (u concurrent access t state (variables) ma race conditions
- So, the first thing we need to do is to identify the fields that may be shared by several threads
- The state of a class involves the fields defined in the class
  - In a nutshell, our goal is to ensure that concurrent manipulation of the class state is race condition free

```
    // class state (variables)
    T s1;
    T s2;
    T s3;
    T s4;
    …

    // class methods
    T m1(…) {…}
    T m2(…) {…}
    T m3(…) {…}
    …
}
```

Escaping


```
class Counter {
      // class state (variables)
      int i=0;

      // class methods
      public synchronized void inc(){i++;}
}
```

Is the class **Counter** thread-safe?

```
class Counter {
      // class state (variables)
      int i=0;

      // class methods
      public synchronized void inc(){i++;}
}
```

Escaping

Is the class `Counter` thread-safe?

```
class Counter {
      // class state (variables)
      int i=0;

      // class methods
      public synchronized void inc(){i++;}
}
```

```
// program using Counter

Counter c = new Counter();
new Thread(() -> {
      c.inc();
}).start();

new Thread(() -> {
      c.i++; // escaped the lock in inc()
}).start();
```

Escaping

- It is important to not expose shared state variables
- Otherwise, threads may use them without proper locking
  - Thus, we allow several threads in the critical section

```
class Counter {
      // class state (variables)
      int i=0;

      // class methods
      public synchronized void inc(){i++;}
}
```

```
// program using Counter

Counter c = new Counter();
new Thread(() -> {
      c.inc();
}).start();

new Thread(() -> {
      c.i++; // escaped the lock in inc()
}).start();
```

Escaping

- It is important to not expose shared state variables
- Otherwise, threads may use them without proper locking
  - Thus, we allow several threads in the critical section
- Defining all (shared) class state (primitive) variables as private ensures that these variables will only be accessed through public methods.
  - Thus, it is easier to control and reason about concurrent access

```
class Counter {
      // class state (variables)
      int i=0;

      // class methods
      public synchronized void inc(){i++;}
}
```

```
// program using Counter

Counter c = new Counter();
new Thread(() -> {
      c.inc();
}).start();

new Thread(() -> {
      c.i++; // escaped the lock in inc()
}).start();
```

Escaping

No, because, even though the inc method is protected by a lock, variable i is public. Consequently, it may be accessed by many threads without being protected by a lock

- It is important
  shared state va
- Otherwise, threads may use them without proper locking
  - Thus, we allow several threads in the critical section
- Defining all (shared) class state (primitive) variables as private ensures that these variables will only be accessed through public methods.
  - Thus, it is easier to control and reason about concurrent access

```
    // class methods
    public synchronized void inc(){i++;}
}
```

```
// program using Counter

Counter c = new Counter();
new Thread(() -> {
    c.inc();
}).start();

new Thread(() -> {
    c.i++; // escaped the lock in inc()
}).start();
```

Escaping


```
class IntArrayList {
    // class state
    private List<Integer> a = new ArrayList<Integer>();

    public synchronized void set(Integer index, Integer elem)
        {  a.set(index,elem); }

    public synchronized List<Integer> get() { return a; }
}
```

Is the class `IntArrayList` thread-safe?


```
class IntArrayList {
    // class state
    private List<Integer> a = new ArrayList<Integer>();

    public synchronized void set(Integer index, Integer elem)
        {  a.set(index,elem); }

    public synchronized List<Integer> get() { return a; }
}
```

Is the class **`IntArrayList`** thread-safe?

```
class IntArrayList {
    // class state
    private List<Integer> a = new ArrayList<Integer>();

    public synchronized void set(Integer index, Integer elem)
        {  a.set(index,elem); }

    public synchronized List<Integer> get() { return a; }
}
```

```
IntArrayList array = new IntArrayList();
new Thread(() -> {
    array.set(0,1); // access state with lock
}).start();
new Thread(() -> {
    array.get().set(0,42); // access state without locks
}).start();
```

Escaping

- Remember that when a method returns an object, we get a *reference* to that object

- Therefore, even if obtain the reference using locks, later we can modify the content of the object without locks

```
class IntArrayList {
    // class state
    private List<Integer> a = new ArrayList<Integer>();

    public synchronized void set(Integer index, Integer elem)
        {  a.set(index,elem); }

    public synchronized List<Integer> get() { return a; }
}
```

```
IntArrayList array = new IntArrayList();
new Thread(() -> {
    array.set(0,1); // access state with lock
}).start();
new Thread(() -> {
    array.get().set(0,42); // access state without locks
}).start();
```

Escaping

- Remember that when a method returns an object, we get a *reference* to that object

- Therefore, even if obtain the reference using locks, later we can modify the content of the object without locks

```
class IntArrayList {
    // class state
    private List<Integer> a = new ArrayList<Integer>();

    public synchronized void set(Integer index, Integer elem)
        {  a.set(index,elem); }

    public synchronized List<Integer> get() { return a; }
}
```

```
IntArrayList array = new IntArrayList();
new Thread(() -> {
    array.set(0,1); // access state with lock
}).start();
new Thread(() -> {
    array.get().set(0,42); // access state without locks
}).start();
```

Is this ***program*** thread-safe?

Escaping

No, because

- Remember th when a method retu s an object, we g a *reference* t that object
- Therefore, even if obtain the reference using locks, later we can modify the content of the object without locks

```
class IntArrayList {
    // class state
    private List<Integer> a = new ArrayList<Integer>();

    public synchronized void set(Integer index, Integer elem)
        {  a.set(index,elem); }

    public synchronized List<Integer> get() { return a; }
}
```

```
IntArrayList array = new IntArrayList();
new Thread(() -> {
    array.set(0,1); // access state with lock
}).start();
new Thread(() -> {
    array.get().set(0,42); // access state without locks
}).start();
```

No, because the value of array[0] after the execution of IntArrayList.set(0,1) may be equal 42. Note that the array.get().set() uses no lock, so the locking and unlocking in IntArrayList.set does not prevent the data race

Safe publication

- It is important to ensure that <u>initialization *happens-before* publication</u>
  - That is, before making accessible a reference to an object, all its fields must be correctly initialized

Safe publication

- It is important to ensure that initialization *happens-before* publication
  - That is, before making accessible a reference to an object, all its fields must be correctly initialized

```
public class UnsafeLazyInitialization {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null)
            resource = new Resource();
        return resource;
    }
}
```

Safe publication

- It is important to ensure that <u>initialization *happens-before* publication</u>
  - That is, before making accessible a reference to an object, all its fields must be correctly initialized

```
public class UnsafeLazyInitialization {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null)
            resource = new Resource();
        return resource;
    }
}
```

Is this class thread-safe?

Safe publication

- It is important to ensure that <u>initialization *happens-before* publication</u>
  - That is, before making accessible a reference to an object, all its fields must be correctly initialized

```
public class UnsafeLazyInitialization {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null)
            resource = new Resource();
        return resource;
    }
}
```

No, if the expected behaviour of getInstance is to create a Resource object only once. Two threads may run getInstance() at the same time, both get resource==null and both creating an instance of Resource. With the last thread's instance being the value that stays in resource (state of the class).

Object initialization & visibility

- Visibility issues may appear during initialization of objects

```
public class UnsafeInitialization {
    private int x;
    private Object o;
    public UnsafeInitialization() {
        x = 42;
        o = new Object();
    }
}
```

Object initialization & visibility

- Visibility issues may appear during initialization of objects


```
public class UnsafeInitialization {
    private int x;
    private Object o;
    public UnsafeInitialization() {
        x = 42;
        o = new Object();
    }
}
```

Object initialization & visibility

- Visibility issues may appear during initialization of objects


```
public class UnsafeInitialization {
    private int x;
    private Object o;
    public UnsafeInitialization() {
        x = 42;
        o = new Object();
    }
}
```

- For the thread executing the constructor, there are no visibility issues, but if a reference to an instance of UnsafeInitialization object is accessible to another thread, it might not see **`x==42`** or **`o`** completely initialized

Object initialization & visibility

- We can address visibility issues during initialization as follows

```
public class UnsafeInitialization {
    private volatile int x;
    private final Object o;
    public UnsafeInitialization() {
        x = 42;
        o = new Object();
    }
}
```

Object initialization & visibility

- We can address visibility issues during initialization as follows

For primitive types, we can:

- Declare them as **`volatile`**
- Declare them as **`final`** (only works if the content is never modified)
- Initialize as the default value: 0. (only works if the default value is acceptable)
- Use corresponding atomic class from Java standard library: **`AtomicInteger`**

```
public class UnsafeInitialization {
    private volatile int x;
    private final Object o;
    public UnsafeInitialization() {
        x = 42;
        o = new Object();
    }
}
```

Object initialization & visibility

- We can address visibility issues during initialization as follows

For primitive types, we can:

- Declare them as **`volatile`**
- Declare them as **`final`** (only works if the content is never modified)
- Initialize as the default value: 0. (only works if the default value is acceptable)
- Use corresponding atomic class from Java standard library: **`AtomicInteger`**

```
public class UnsafeInitialization {
    private volatile int x;
    private final Object o;
    public UnsafeInitialization() {
        x = 42;
        o = new Object();
    }
}
```

For complex objects, we can:

- Declare them as **`final`**
- Initialize as the default value: null. (only works if the default value is acceptable)
- Use the **`AtomicReference`** class

Object initialization & visibility

- We can address visibility issues during initialization as follows

For primitive types, we can:
- Declare them as **`volatile`**
- Declare them as **`final`** (only works if the content is never modified)
- Initialize as the default value: 0. (only works if the default value is acceptable)
- Use corresponding atomic class from Java standard library: **`AtomicInteger`**

```
public class UnsafeInitialization {
    private volatile int x;
    private final Object o;
    public UnsafeInitialization() {
        x = 42;
        o = new Object();
    }
}
```

For complex objects, we can:
- Declare them as **`final`**
- Initialize as the default value: null. (only works if the default value is acceptable)
- Use the **`AtomicReference`** class

Why do these solutions solve visibility issues?

Object initialization & visibility

- We can address visibility issues during initialization as follows

For primitive types, we can:
- Declare them as **`volatile`**
- Declare them as **`final`** (only works if the content is never modified)
- Initialize as the default value: 0. (only works if the default value is acceptable)
- Use corresponding atomic class from Java standard library: **`AtomicInteger`**

```
public class UnsafeInitialization {
    private volatile int x;
    private final Object o;
    public UnsafeInitialization() {
        x = 42;
        o = new Object();
    }
}
```

For complex objects, we can:
- Declare them as **`final`**
- Initialize as the default value: null. (only works if the default value is acceptable)
- Use the **`AtomicReference`** class

For x, because volatile variables are flushed to main memory for every write (so all threads read the same values). For o, because the JVM ensures visibility of complex objects declared as final

Object initialization & visibility

- The previous suggestions ensure safe publication because:
  - They established a *happens-before* relation between initialization and access the object's reference (publication)
    - *A write to a volatile field happens-before every subsequent read of that field.*
    - *The default initialization (zero, false, or null) of any object happens-before any other actions of a program.*
    - *The initialization of a final field happens-before any other actions of a program (after the constructor has finished its execution)*
  - At the JVM level, the reason is that
    - **`final`** fields cannot be cached or reordered during initialization
    - All fields are initialized with default values during class loading
    - writes on **`volatile`** are flushed to main memory and reordered (during initialization)

Object initialization & visibility

- The previous suggestions ensure safe publication because:
  - They established a *happens-before* relation between initialization and access the object's reference (publication)
    - *A write to a volatile field happens-before every subsequent read of that field.*
    - *The default initialization (zero, false, or null) of any object happens-before any other actions of a program.*
    - *The initialization of a final field happens-before any other actions of a program (after the constructor has finished its execution)*

Defined by us from the JLS explanation. You can use for exercises in this course.

  - At the JVM level, the reason is that
    - **`final`** fields cannot be cached or reordered during initialization
    - All fields are initialized with default values during class loading
    - writes on **`volatile`** are flushed to main memory and reordered (during initialization)

Object initialization & visibility

- The previous suggestions ensure safe publication because:
  - They established a *happens-before* relation between initialization and access the object's reference (publication)
    - *A write to a volatile field happens-before every subsequent read of that field.*
    - *The default initialization (zero, false, or null) of any object happens-before any other actions of a program.*
    - *The initialization of a final field happens-before any other actions of a program (after the constructor has finished its execution)*

Defined by us from the JLS explanation. You can use for exercises in this course.

If the constructor of the class leaks a reference of the object being constructed before it has completed its execution, then there is no happen-before relation with the accesses to final field

  - At the JVM level, the reason is that
    - `final` fields cannot be cached or reordered during initialization
    - All fields are initialized with default values during class loading
    - writes on `volatile` are flushed to main memory and reordered (during initialization)

Object initialization & visibility

NOTE: For clarity and simplicity, up to now, we did not take initialization concerns into account. But from now on we will.

- The previous suggestions ensure safe publication because:
  - They established a *happens-before* relation between initialization and access the object's reference (publication)
    - *A write to a volatile field happens-before every subsequent read of that field.*
    - *The default initialization (zero, false, or null) of any object happens-before any other actions of a program.*
    - *The initialization of a final field happens-before any other actions of a program (after the constructor has finished its execution)*

Defined by us from the JLS explanation. You can use for exercises in this course.

If the constructor of the class leaks a reference of the object being constructed before it has completed its execution, then there is no happen-before relation with the accesses to final field

  - At the JVM level, the reason is that
    - `final` fields cannot be cached or reordered during initialization
    - All fields are initialized with default values during class loading
    - writes on `volatile` are flushed to main memory and reordered (during initialization)

Immutability

- An immutable object is one whose state cannot be changed after initialization
  - You can think of it as a constant
  - The **`final`** keyword in Java prevents modification of fields
    - Remember that variables assigned to an object only hold a reference to the object

- A immutable class is one whose instances are immutable objects

Immutability

- An immutable object is one whose state cannot be changed after initialization
  - You can think of it as a constant
  - The **`final`** keyword in Java prevents modification of fields
    – Remember that variables assigned to an object only hold a reference to the object

- A immutable class is one whose instances are immutable objects

Are immutable classes thread-safe?

Immutability

- An immutable object is one whose state cannot be changed after initialization
  - You can think of it as a constant
  - The **`final`** keyword in Java prevents modification of fields
    – Remember that variables assigned to an object only hold a reference to the object

- A immutable class is one whose instances are immutable objects

If they are safely published (i.e., without visibility issues during publication), then *yes*. This is because fields can only be read.

Immutable class & `final`

Immutable class & **`final`**

Does defining all fields as **`final`** ensure that the class is immutable?

Immutable class & `final`

Does defining all fields as `final` ensure that the class is immutable?

If in a class, no fields are defined as `final`, is it possible to make it immutable?

Immutable class & `final`

No, complex objects may still be updated, e.g., we can add elements to a list declared as final.

Yes, if we ensure that those fields cannot be accessed, and have methods that only read them.

Immutability

- To ensure thread-safety of immutable classes you simply need to make sure:
  - No fields can be modified after publication
  - Objects are safely published
  - Access to inner mutable object do not escape

Immutability

- To ensure thread-safety of immutable classes you simply need to make sure:
  - No fields can be modified after publication
  - Objects are safely published
  - Access to inner mutable object do not escape

```
public final class ThreeStooges {
  private final Set<String> stooges = new HashSet<String>();

  public ThreeStooges () {
    stooges.add("Moe");
    stooges.add("Larry");
    stooges.add("Curly");
  }

  public Boolean isStooge(String name) {
    return stooges.contains(name)
  }
}
```

Goetz p. 47

Immutability

- To ensure thread-safety of immutable classes you simply need to make sure:
  - No fields can be modified after publication
  - Objects are safely published
  - Access to inner mutable object do not escape

```
public final class ThreeStooges {
  private final Set<String> stooges = new HashSet<String>();

  public ThreeStooges () {
    stooges.add(“Moe”);
    stooges.add(“Larry”);
    stooges.add(“Curly”);
  }

  public Boolean isStooge(String name) {
    return stooges.contains(name)
  }
}
```

Why is this class thread-safe?
(tip: there are 3 main reasons)

Goetz p. 47

Immutability

- To ensure thread-safety o[...] sure:
  - No fields can be modi[...]
  - Objects are safely pub[...]
  - Access to inner mutable object do not escape

1. The state (stooges) is declared as final so it is safely published.
2. The state is declared as private and no method returns a reference to stooges, so there is no way to access the field other than using the method isStooge()
3. The method isStooge() does not modify the state, only reads it

```
public final class ThreeStooges {
  private final Set<String> stooges = new HashSet<String>();

  public ThreeStooges () {
    stooges.add("Moe");
    stooges.add("Larry");
    stooges.add("Curly");
  }

  public Boolean isStooge(String name) {
    return stooges.contains(name)
  }
}
```

Goetz p. 47

Mutual exclusion

- Whenever shared mutable state is accessed by several threads is must be protected by locks

Mutual exclusion

- Whenever shared <u>mutable</u> state is accessed by several threads is must be protected by locks

Are Monitors a thread-safe class?
(when implemented as a class in OO languages)

Mutual exclusion

- Whenever shared mutable state is accessed by several threads is must be protected by locks

Are Monitors a thread-safe class?
(when implemented as a class in OO languages)

Is it always necessary to use locks in the methods of thread-safe classes?

Mutual exclusion

- Whenever shared <u>mutable</u> state is accessed by several threads is must be protected by locks

Yes, if implemented according to the specification we set forth in lecture 2, and they are safely published

No, e.g., the class ThreeStooges two slides ago. Locks are necessary if the state is written or written and read by different threads

Thread-safe classes

- To analyse thread-safe in a class, we must identify/consider:
  - Identify the class state
  - Make sure that mutable class state does not escape
  - Ensure safe publication
  - Whenever possible define class state as immutable
  - If class state must be mutable, ensure mutual exclusion

Interesting section (4.5) on documenting synchronization in Goetz. Unfortunately, not widespread.

Instance confinement

- *Instance confinement* refers to encapsulating access to a thread-unsafe object into a thread-safe class

Instance confinement

- *Instance confinement* refers to encapsulating access to a thread-unsafe object into a thread-safe class

```
public class PersonSet {
  private final Set<Person> mySet = new HashSet<Person>();

  public synchronized void addPerson (Person p)  {
    mySet.add(p);
  }

  public synchronized boolean contains(Person p)  {
    return mySet.contains(p);
  }
}
```

Goetz p. 59

Instance confinement

- *Instance confinement* refers to encapsulating access to a thread-unsafe object into a thread-safe class

```
public class PersonSet {
  private final Set<Person> mySet = new HashSet<Person>();

  public synchronized void addPerson (Person p) {
    mySet.add(p);
  }

  public synchronized boolean contains(Person p) {
    return mySet.contains(p);
  }
}
```

Goetz p. 59

Why is this class thread-safe?

Instance confinement

- *Instance confinement* refers to encapsulating access to a thread-unsafe object into a thread-safe class

```
public class PersonSet {
  private final Set<Person> mySet = new HashSet<Person>();

  public synchronized void addPerson (Person p) {
    mySet.add(p);
  }

  public synchroniz
    return mySet.co
  }
}
```

The state cannot escape (because is declared as private), the object is safely published (mySet declared as final), mutual exclusion is ensured via intrinsic locks in all methods.

Synchronized collections

- Java's standard library provides a method to convert ordinary collections in to "synchronized" collections
  - **`synchronizedCollection(Collection<T> c)`**, **`synchronizedList(List<T> l)`**, **`synchronizedSet(Set<T> s)`**, …, **`synchronizedXXX(XXX<T> x)`** with **`XXX`** a Java collection.
  - Internally, these methods turn all the methods in the collection into synchronized
    - That is, they use the instance lock

Synchronized collections

- Java’s standard library provides a method to convert ordinary collections in to “synchronized” collections
  - `synchronizedCollection(Collection<T> c)`, `synchronizedList(List<T> l)`, `synchronizedSet(Set<T> s)`, …, `synchronizedXXX(XXX<T> x)` with **`XXX`** a Java collection.
  - Internally, these methods turn all the methods in the collection into synchronized
    – That is, they use the instance lock

Are synchronized collections thread-safe?

Synchronized collections

- Java’s standard library provides a method to convert ordinary collections in to “synchronized” collections
  - `synchronizedCollection(Collection<T> c)`, `synchronizedList(List<T> l)`, `synchronizedSet(Set<T> s)`, …, `synchronizedXXX(XXX<T> x)` with **XXX** a Java collection.
  - Internally, these methods turn all the methods in the collection into synchronized
    – That is, they use the instance lock

Are synchronized collections thread-safe?

Let’s look at the Javadoc (https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#synchronizedList-java.util.List-)

Synchronized collections

- Java's standard library provides a method to convert ordinary collections in to "synchronized" collections
  - `synchronizedCollection(Collection<T> c)`, `synchronizedList(List<T> l)`, `synchronizedSet(Set<T> s)`, ..., `synchronizedXXX(XXX<T> x)` with **`XXX`** a Java collection.
  - Internally, these methods turn all the methods in the collection into synchronized
    - That is, t

Yes. In class I said that this depends on whether the method subList() uses the same lock as the original list. This is indeed the case---you can verify this yourself in by looking into the implementation of subList in the JDK.

Let's look at the Javadoc
(https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#synchronizedList-java.util.List-)

p only accesses thread-safe <u>classes</u> $\not\Rightarrow$ p is a thread-safe <u>program</u>

p only accesses thread-safe classes ⇏ p is a thread-safe program


```
List<Integer> l = new ArrayList<Integer>();
List<Integer> lSync = Collections.synchronizedList(l);

…

new Thread(() -> { addIfAbsent(lSync,1); }).start();
new Thread(() -> { addIfAbsent(lSync,1); }).start();

…

public void addIfAbsent(List l, Integer e) {
  if (!l.contains(e))
    l.add(e);
}
```

p only accesses thread-safe classes ⇏ p is a thread-safe program

Is this program thread-safe?

```
List<Integer> l = new ArrayList<Integer>();
List<Integer> lSync = Collections.synchronizedList(l);

…

new Thread(() -> { addIfAbsent(lSync,1); }).start();
new Thread(() -> { addIfAbsent(lSync,1); }).start();

…

public void addIfAbsent(List l, Integer e) {
  if (!l.contains(e))
    l.add(e);
}
```

Remember: *thread-safe class* ⇏ *thread-safe program*

Here are reaching the limits of our definition of thread-safety. To be completely formal we should precisely define what is the expected behaviour of addIfAbsent().

If we take as the expected behaviour of addIfAbsent that the element should only be added in the list if it has not been inserted by any other thread, then this class is not thread-safe. This is because 2 threads could evaluate the if-condition in addIfAbsent to true, and then add the element twice.

```
List<Integer> l = new ArrayList<Integer>();
List<Integer> lSync = Collections.synchronizedList(l);

…

new Thread(() -> { addIfAbsent(lSync,1); }).start();
new Thread(() -> { addIfAbsent(lSync,1); }).start();

…

public void addIfAbsent(List l, Integer e) {
  if (!l.contains(e))
    l.add(e);
}
```

Extending thread-safe classes

- Thread-safe classes may be extended to include compound actions
  - Intuitively, compound actions can be seen multiple method calls or field accesses within a critical section
  - A common examples are: *check-and-set,* iteration, navigation (*contains*)

```
public void addIfAbsent(List l, Integer e) {
  synchronized (l) {
    if (!l.contains(e))
      l.add(e);
  }
}
```

Thread uses the intrinsic lock of a synchronized collection

```
class ThreadSafeList {
  …
  public void synchronized addIfAbsent(T e) {
    if (!l.contains(e))
        l.add(e);
  }
  …
}
```

Thread-safe class is extended with a custom method to perform the action

Other synchronization primitives (synchronizers)

Semaphores

- Semaphores are synchronization primitives that allow at most *c* number of threads in the critical section where *c* is called the *capacity*
  - First introduced by Dijkstra

- A semaphore consists of:
  - An integer capacity (*c*), permits in Java
    - Initial number of threads allowed in the critical section
  - A method **`acquire()`**
    - Checks if $c > 0$, if so, it decrements capacity by one (c--) and allows the calling thread to make progress, otherwise it blocks the thread
    - It is a blocking call
  - A method **`release()`**
    - It checks whether there are waiting threads, if so, it wakes up one of them, otherwise it increases the capacity by one (c++)
    - It is non-blocking

Semaphores

- Semaphores are synchronization primitives that allow at most *c* number of threads in the critical section where *c* is called the *capacity*
  - First introduced by Dijkstra

Semaphores (1968) appear before Monitors (1972)

- A semaphore consists of:
  - An integer capacity (*c*), permits in Java
    - Initial number of threads allowed in the critical section
  - A method **`acquire()`**
    - Checks if $c > 0$, if so, it decrements capacity by one (c--) and allows the calling thread to make progress, otherwise it blocks the thread
    - It is a blocking call
  - A method **`release()`**
    - It checks whether there are waiting threads, if so, it wakes up one of them, otherwise it increases the capacity by one (c++)
    - It is non-blocking

Semaphores

If we set the capacity of a semaphore to 1, does it behave like a lock?

- Semaphores are synchronization primitives that allow at most *c* number of threads in the critical section where *c* is called the *capacity*
  - First introduced by Dijkstra

Semaphores (1968) appear before Monitors (1972)

- A semaphore consists of:
  - An integer capacity (*c*), [permits in Java]()
    - Initial number of threads allowed in the critical section
  - A method **`acquire()`**
    - Checks if $c > 0$, if so, it decrements capacity by one (c--) and allows the calling thread to make progress, otherwise it blocks the thread
    - It is a blocking call
  - A method **`release()`**
    - It checks whether there are waiting threads, if so, it wakes up one of them, otherwise it increases the capacity by one (c++)
    - It is non-blocking

Semaphores

If we set the capacity of a semaphore to 1, does it behave like a lock?

- Semaphores are synchronization primitives that allow at most *c* number of threads in the critical section where *c* is called the

Synchronization primitives that only allow one thread in the critical section are called ***mutex*** (which is short for mutual exclusion)

Semaphores (1968) appear before Monitors (1972)

- An integer capacity (*c*), [permits in Java]
  - Initial number of threads allowed in the critical section
- A method **`acquire()`**
  - Checks if $c > 0$, if so, it decrements capacity by one (c--) and allows the calling thread to make progress, otherwise it blocks the thread
  - It is a blocking call
- A method **`release()`**
  - It checks whether there are waiting threads, if so, it wakes up one of them, otherwise it increases the capacity by one (c++)
  - It is non-blocking

Semaphores

Depends on the behaviour of release. If release increases the capacity of the semaphore when c==1, then no. But if the semaphore does not increase capacity when c==1, then yes.

- Semaphores ar
  number of thre

Synchronization primiti
one thread in the critical section are called ***mutex*** (which is short for mutual exclusion)

Semaphores (1968) appear before Monitors (1972)

- An integer capacity (*c*), [permits in Java]
  - Initial number of threads allowed in the critical section
- A method **`acquire()`**
  - Checks if *c* > 0, if so, it decrements capacity by one (c--) and allows the calling thread to make progress, otherwise it blocks the thread
  - It is a blocking call
- A method **`release()`**
  - It checks whether there are waiting threads, if so, it wakes up one of them, otherwise it increases the capacity by one (c++)
  - It is non-blocking

Semaphores | Intuition

- You can think of a semaphore as a “bouncer” to enter a critical section or to be allowed to used a shared resource

Semaphores | Intuition

- You can think of a semaphore as a “bouncer” to enter a critical section or to be allowed to used a shared resource

Semaphores | Intuition

- You can think of a semaphore as a “bouncer” to enter a critical section or to be allowed to used a shared resource

Semaphores | Intuition

- You can think of a semaphore as a “bouncer” to enter a critical section or to be allowed to used a shared resource

Semaphores | Intuition

- You can think of a semaphore as a “bouncer” to enter a critical section or to be allowed to used a shared resource

Semaphores | Example

- Semaphores are typically used to control the number of threads accessing a resource (here we fix a maximum 5 readers and writers)

```
ReadWriteMonitor m = new ReadWriteMonitor();
Semaphore semReaders = new Semaphore(5,true);
Semaphore semWriters = new Semaphore(5,true);
for (int i = 0; i < 10; i++) {
    // start a reader
    new Thread(() -> {
            m.readLock();
            semReaders.acquire();
            // read
            semReaders.release();
            m.readUnlock();
    }).start();

    // start a writer
    new Thread(() -> {
            m.writeLock();
            semWriters.acquire();
            // write
            semWriters.acquire();
            m.writeUnlock();
    }).start();
}
```

Java semaphores have a fair flag so that their entry queue prioritizes the longest waiting thread

Semaphores | Example

- Semaphores are typically used to control the number of threads accessing a resource (here we fix a maximum 5 readers and writers)

```
ReadWriteMonitor m = new ReadWriteMonitor();
Semaphore semReaders = new Semaphore(5,true);
Semaphore semWriters = new Semaphore(5,true);
for (int i = 0; i < 10; i++) {
    // start a reader
    new Thread(() -> {
        m.readLock();
        semReaders.acquire();
        // read
        semReaders.release();
        m.readUnlock();
    }).start();

    // start a writer
    new Thread(() -> {
        m.writeLock();
        semWriters.acquire();
        // write
        semWriters.acquire();
        m.writeUnlock();
    }).start();
}
```

Java semaphores have a fair flag so that their entry queue prioritizes the longest waiting thread

Does the semaphore make any difference for writers?

See **`ReadersWritersSemaphore.java`**

Semaphores | Example

- Semaphores are typically used to control the number of threads accessing a resource (here we fix a maximum 5 readers and writers)

```
ReadWriteMonitor m = new ReadWriteMonitor();
Semaphore semReaders = new Semaphore(5,true);
Semaphore semWriters = new Semaphore(5,true);
for (int i = 0; i < 10; i++) {
    // start a reader
    new Thread(() -> {
            m.readLock();
            semReaders.acquire();
            // read
            semReaders.release();
            m.readUnlock();
    }).start();

    // start a writer
    new Thread(() -> {
            m.writeLock();
            semWriters.acquire();
            // write
            semWriters.acquire();
            m.writeUnlock();
    }).start();
}
```

Java semaphores have a fair flag so that their entry queue prioritizes the longest waiting thread

Do we need a semaphore to impose this constraint or can we implement it in the monitor?

Does the semaphore make any difference for writers?

See **ReadersWritersSemaphore.java**

Semaphores | Example

- Semaphores are typically used to controlled the number of threads that can access a resource

```
ReadWriteMonitor m = new ReadWriteMonitor();
Semaphore semReaders = new Semaphore(5,true);
Semaphore semWriters = new Semaphore(5,true);
for (int i = 0; i < 10; i++) {
    // start a reader
    new Thread(() -> {
            semReaders.acquire();
            m.readLock();
            // read
            m.readUnlock();
            semReaders.release();
    }).start();

    // start a writer
    new Thread(() -> {
            semWriters.acquire();
            m.writeLock();
            // write
            m.writeUnlock();
            semWriters.acquire();
    }).start();
}
```

Java semaphores have a fair flag so that their entry queue prioritizes the longest waiting thread

See **`ReadersWritersSemaphore.java`**

Semaphores | Example

- Semaphores are typically used to controlled the number of threads that can access a resource

```
ReadWriteMonitor m = new ReadWriteMonitor();
Semaphore semReaders = new Semaphore(5,true);
Semaphore semWriters = new Semaphore(5,true);
for (int i = 0; i < 10; i++) {
    // start a reader
    new Thread(() -> {
          semReaders.acquire();
          m.readLock();
          // read
          m.readUnlock();
          semReaders.release();
    }).start();

    // start a writer
    new Thread(() -> {
          semWriters.acquire();
          m.writeLock();
          // write
          m.writeUnlock();
          semWriters.acquire();
    }).start();
}
```

Java semaphores have a fair flag so that their entry queue prioritizes the longest waiting thread

There is no difference as the monitor only allows only one writer at a time.

See `ReadersWritersSemaphore.java`

Semaphores | Example

- Semaphores are typically used to controlled the number of threads that can access a resource

```
ReadWriteMonitor m = new ReadWriteMonitor();
Semaphore semReaders = new Semaphore(5,true);
Semaphore semWriters = new Semaphore(5,true);
for (int i = 0; i < 10; i++) {
    // start a reader
    new Thread(() -> {
        semReaders.acquire();
        m.readLock();
        // read
        m.readUnlock();
        semReaders.release();
    }).start();

    // start a writer
    new Thread(() -> {
        semWriters.acquire();
        m.writeLock();
        // write
        m.writeUnlock();
        semWriters.acquire();
    }).start();
}
```

Java semaphores have a fair flag so that their entry queue prioritizes the longest waiting thread

No, it can be introduced in the monitor condition

There is no difference as the monitor only allows only one writer at a time.

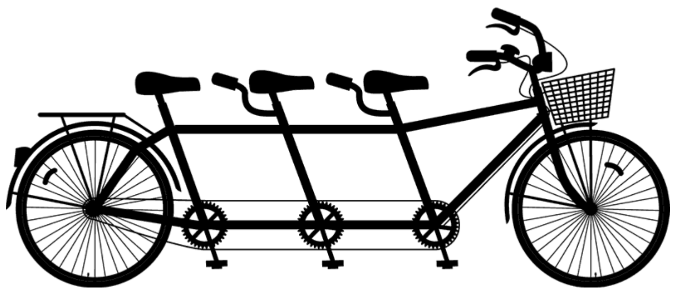
See **`ReadersWritersSemaphore.java`**

Barriers

- *Barriers* are synchronization primitives used to wait until several thread reach some point in their computation

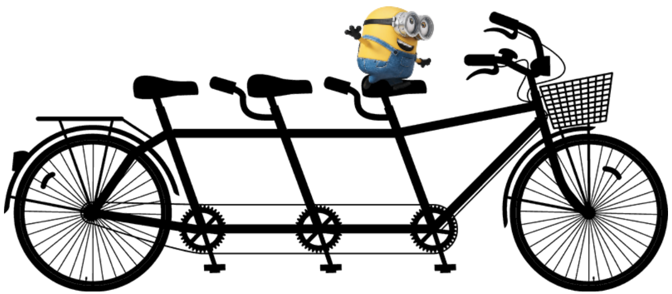
Barriers | Intuition

- *Barriers* are synchronization primitives used to wait until several thread reach some point in their computation



Barriers | Intuition

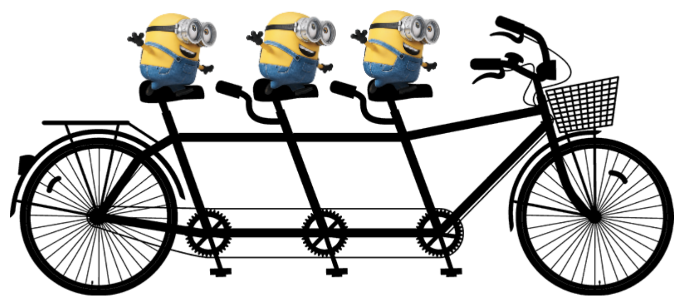
- *Barriers* are synchronization primitives used to wait until several thread reach some point in their computation



Barriers | Intuition

- *Barriers* are synchronization primitives used to wait until several thread reach some point in their computation



Barriers

- *Barriers* are synchronization primitives used to wait until several thread reach some point in their computation

- Barriers consists of
  - A number *parties* to wait for
  - A method **`await()`**
    – If the number of waiting threads is less than *parties*, then the calling thread blocks, otherwise all waiting threads wake up and the calling thread is allowed to make progress

- Java includes the class **`CyclicBarrier`**
  - After *parties* called **`await()`**, then the state is reset and the barrier behaves as initially

Barrier Example | Parallel initialization

- Several threads are used to initialize an array (each a different position), the barrier is used for threads to know when the initialization is finished
  - This example is a bit artificial, but it illustrates the use of barriers.

```
int parties         = 10;
CyclicBarrier cb    = new CyclicBarrier(parties);
int[] shared_array = new int[parties];
…
for (int i = 0; i < parties; i++) {
  new SetterClass(i).start();
}
…
public class SetterClass extends Thread {
  int index;
  public SetterClass(int index) {this.index = index;}

  public void run() {
    shared_array[index] = index+1;
    cb.await();
    // After this point the array is initialized and it is safe to read it
  }
}
```

Barrier Example | Parallel initialization

- Several threads are used to initialize an array (each a different position), the barrier is used for threads to know when the initialization is finished
  - This example is a bit artificial, but it illustrates the use of barriers.

See `BarrierExample.java`

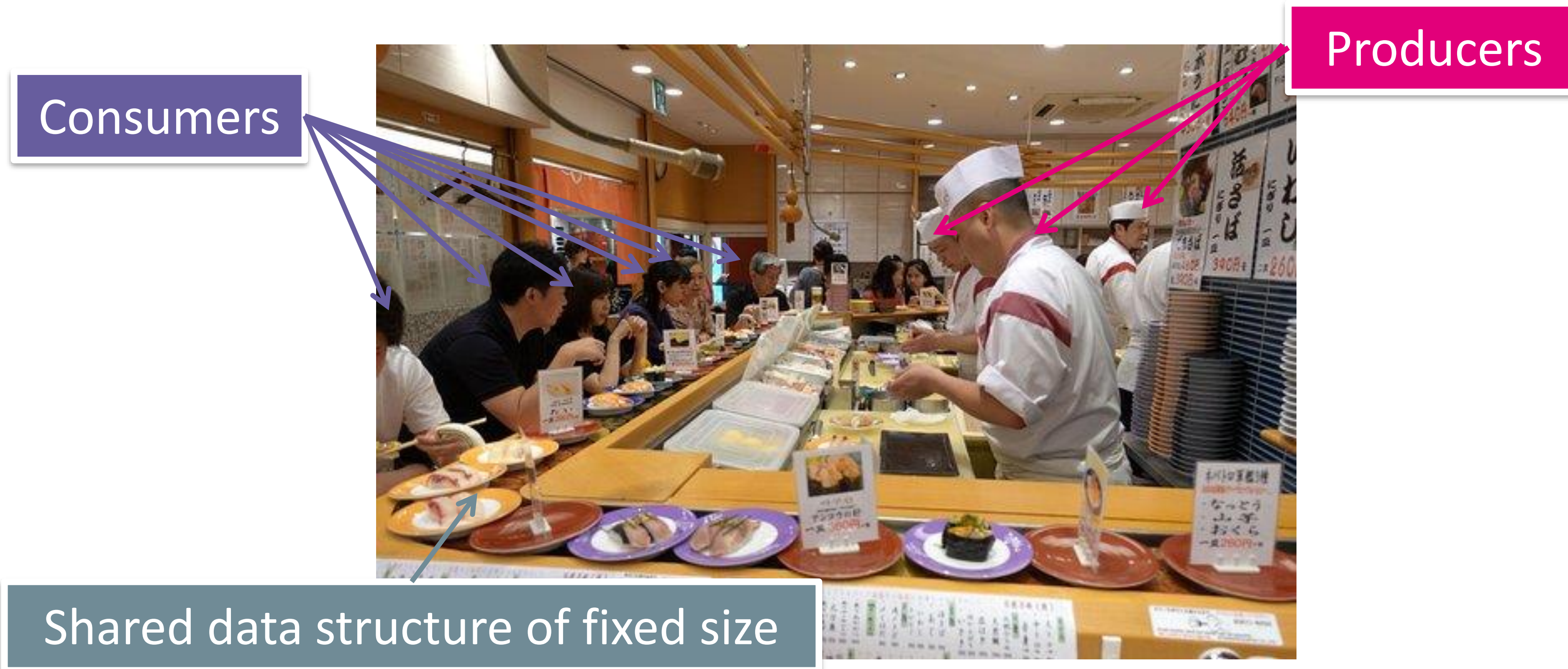
```
int parties        = 10;
CyclicBarrier cb   = new CyclicBarrier(parties);
int[] shared_array = new int[parties];
…
for (int i = 0; i < parties; i++) {
  new SetterClass(i).start();
}
…
public class SetterClass extends Thread {
  int index;
  public SetterClass(int index) {this.index = index;}

  public void run() {
    shared_array[index] = index+1;
    cb.await();
    // After this point the array is initialized and it is safe to read it
  }
}
```

Producer-consumer problem

- Consider a shared data structure of fixed size from which threads may add and remove elements
- *Producer* threads may add elements to the structure as long as it is not full
  - If the structure is full and a producer tries to add an element, it must block until there an element is removed
- *Consumer* threads remove elements to the structure as long as it is not empty
  - If the structure is empty and a consumer tries to remove an element, then it must block until an element is added
- A good solution to the problem must be deadlock free and (possibly) starvation free

Producer-consumer problem | Intuition

- Perhaps more intuitive example



Producer-consumer problem

- The producer-consumer problem appears in many multi-threaded situates
  - Handling access to a shared bounded data structure
  - Controlling access to limited computational resources
    – E.g., thread pools
  - Asynchronous I/O operations
    – External devices may at as producers providing data to the system (keyboard, mouse, etc…), or consumer obtaining tasks to perform (IoT devices)

Agenda

- Definitions of thread-safety
  - Classes
  - Programs
- Safe publication
- Immutability
- Instance confinement
- Synchronization primitives (synchronizers)
  - Semaphores
  - Barriers
- Producer-consumer problem