

Java Threads

Outline

- **Creating a Java Thread**
- Synchronized Keyword
- Wait and Notify

Creating a Java thread

- Java threads are a way to have different parts of your program running at the same time.
- For example, you can create an application that accepts input from different users at the same time, each user handled by a thread.
- Also, most networking applications involve threads
 - you would need to create a thread that would wait for incoming messages while the rest of your program handles your outgoing messages.

Creating a Java thread

- There are two ways to create a java thread
 - Extending the Thread class
 - Implementing the runnable interface

Basic threads

- We will create a thread that simply prints out a number 500 times in a row.

```
class MyThread extends Thread {  
    int i;  
    MyThread(int i) {  
        this.i = i;  
    }  
    public void run() {  
        for (int ctr=0; ctr < 500; ctr++) {  
            System.out.print(i);  
        }  
    }  
}
```

Basic threads

- To show the difference between parallel and non-parallel execution, we have the following executable MyThreadDemo class

```
class MyThreadDemo {  
    public static void main(String args[]) {  
        MyThread t1 = new MyThread(1);  
        MyThread t2 = new MyThread(2);  
        MyThread t3 = new MyThread(3);  
        t1.run();  
        t2.run();  
        t3.run();  
        System.out.print("Main ends");  
    }  
}
```

Basic threads

- Upon executing MyThreadDemo, we get output that looks like this

```
$ java MyThreadDemo
```

```
1111111111...22222222.....33333.....Main ends
```

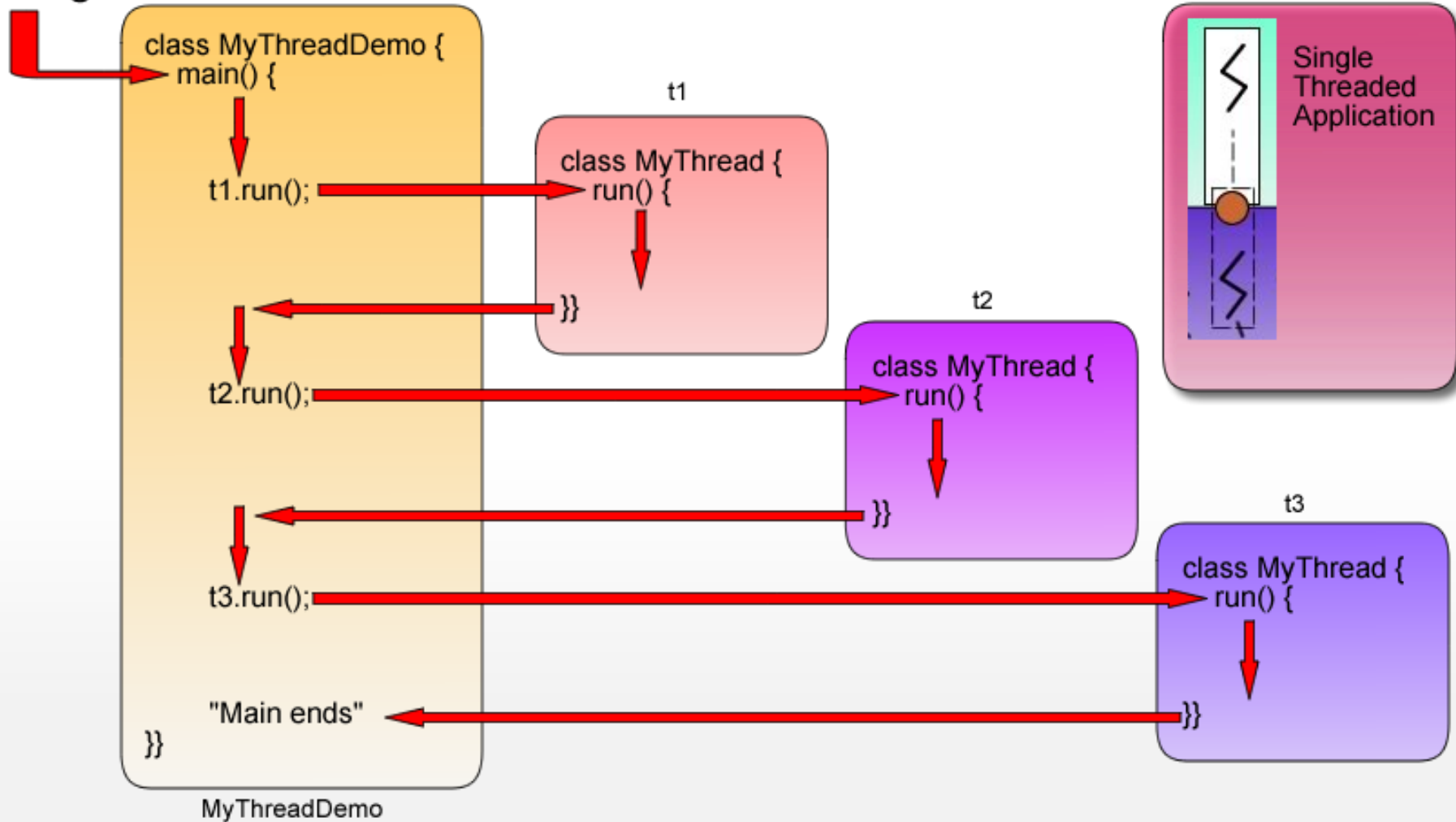
- As can be seen, our program first executed t1's run method, which prints 500 consecutive 1's.
- After that t2's run method, which prints consecutive 2's, and so on.
- Main ends appears at the last part of the output as it is the last part of the program.

Basic threads

- What happened was serial execution, no multithreaded execution occurred
 - This is because we simply called `MyThread's run()` method

Basic threads

Program Flow



Basic threads

- To start parallel execution, we call MyThread's start() method, which is built-in in all thread objects.

```
class MyThreadDemo {  
    public static void main(String args[]) {  
        MyThread t1 = new MyThread(1);  
        MyThread t2 = new MyThread(2);  
        MyThread t3 = new MyThread(3);  
        t1.start();  
        t2.start();  
        t3.start();  
        System.out.print("Main ends");  
    }  
}
```

Basic threads

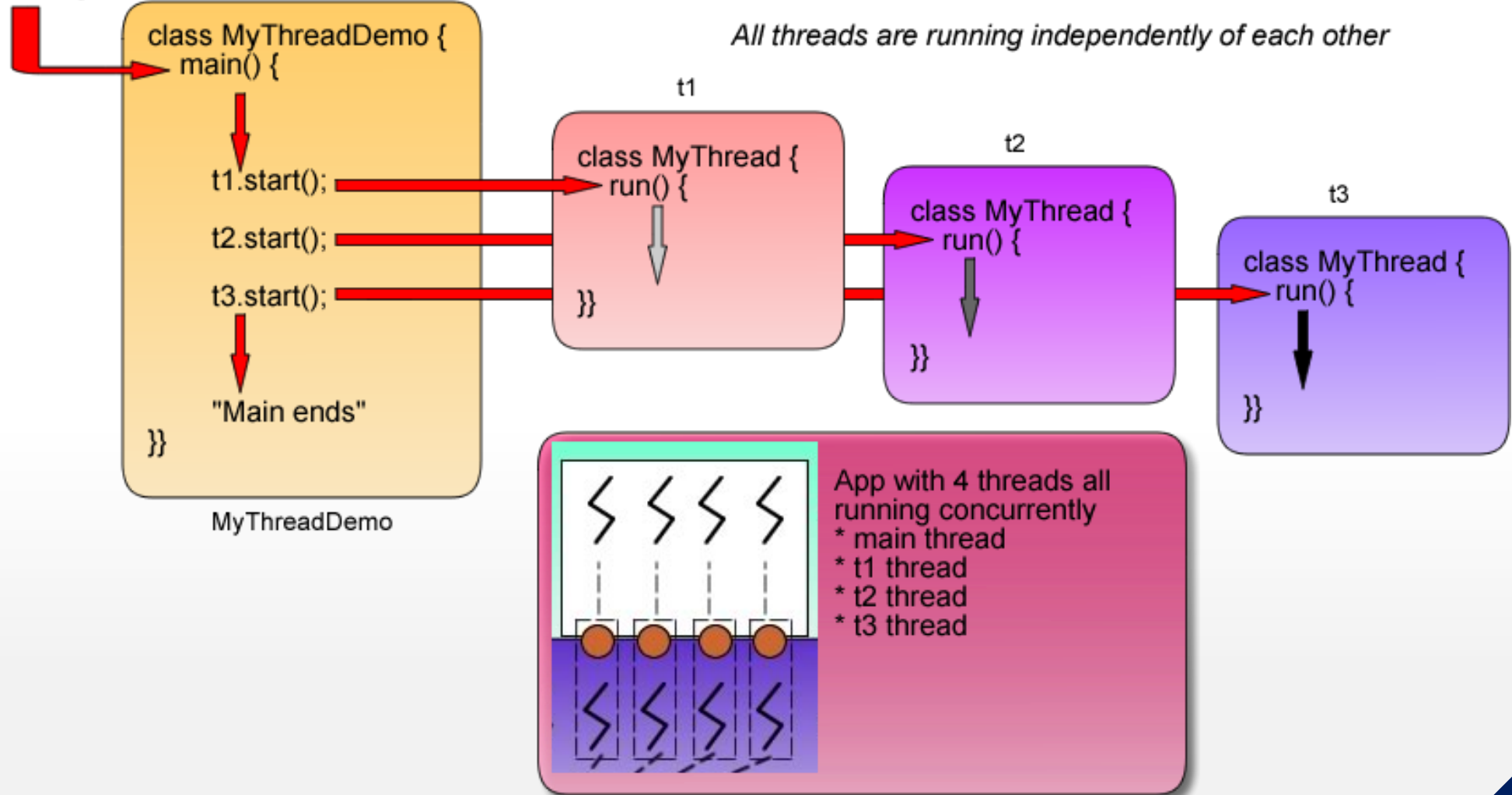
- When we run MyThreadDemo we can definitely see that three run methods are executing at the same time

```
> java MyThreadDemo
```

```
11111111112233112232112233331111Main ends111333222...
```

Basic threads

Program Flow



Basic threads

- Note the appearance of the "Main ends" string in the middle of the output sequence

```
> java MyThreadDemo
```

```
111111111122331122321122333331111Main ends111333222...
```

- This indicates that the main method has already finished executing while thread 1, thread 2 and thread 3 are still running.

Basic threads

- Running a main method creates a thread.
 - Normally, your program ends when the main thread ends.
- However, creating and then running a thread's start method creates a whole new thread that executes its run() method independent of the main method.

Implementing runnable

- Another way to create a thread is to implement the Runnable interface.
- This may be desired if you want your thread to extend another class.
 - By extending another class, you will be unable to extend class thread as Java classes can only extend a single class.
 - However, a class can implement more than one interface.

Implementing runnable

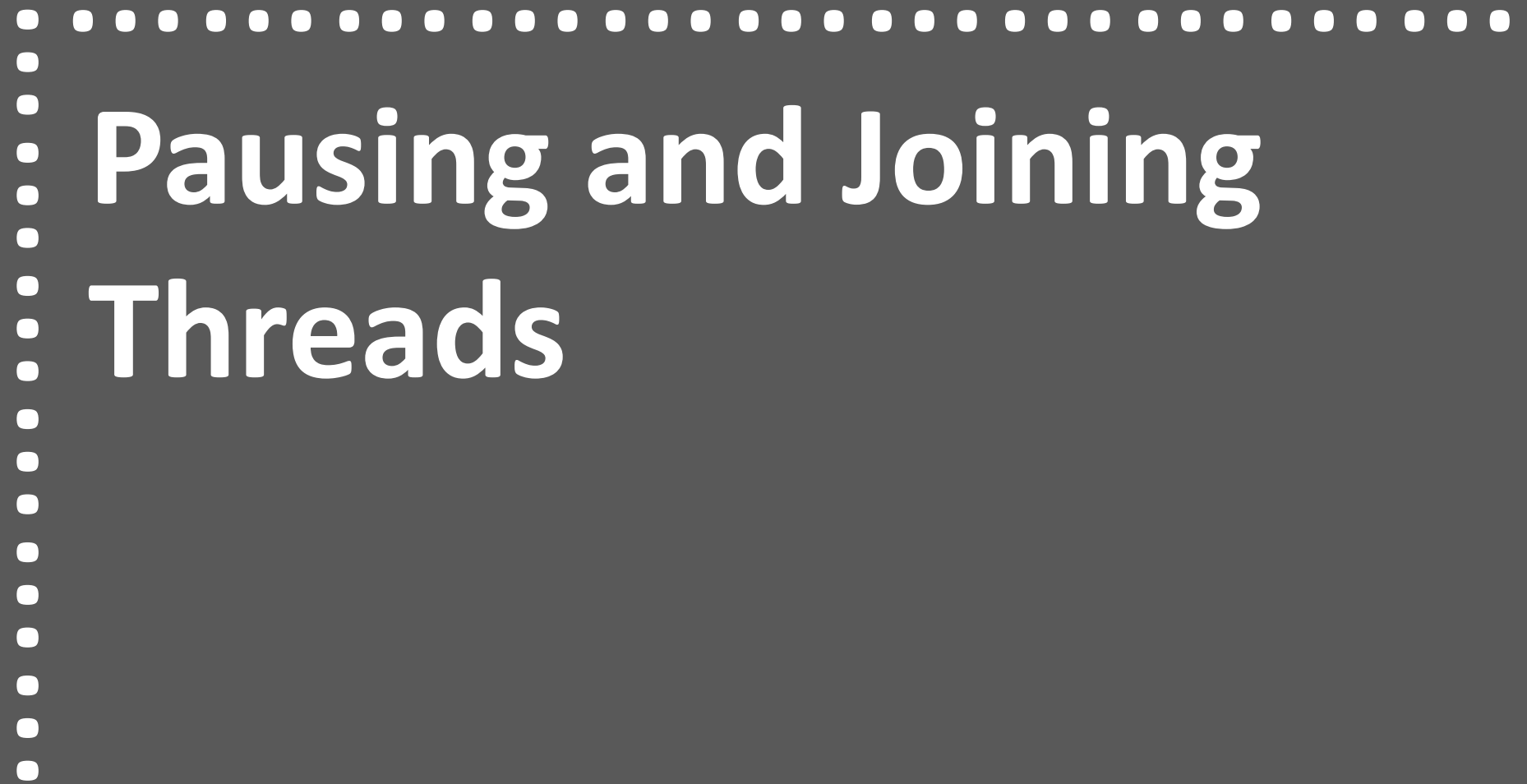
- The following is our MyThread class created by implementing the runnable interface.

```
class MyThread implements Runnable {  
    ... <thread body is mostly the same>  
}
```

- Classes that implement Runnable are instantiated in a different way.
 - Thread t1 = new Thread(new MyThread(1));

Implementing runnable

- Note that implementing an interface requires you to define all the methods in the interface
- For the Runnable interface, you are required in your implementing class to define the run() method or your program will not run.

A decorative border made of white dots forms a rectangular frame around the text. The border is composed of a top horizontal line, a right vertical line, and a left vertical line. The bottom horizontal line is not present.

Pausing and Joining Threads

Pausing threads

- Threads can be paused by the sleep() method.
- For example, to have MyThread pause for half a second before it prints the next number, we add the following lines of code to our for loop.

```
for (int ctr=0; ctr < 500; ctr++) {  
    System.out.print(i);  
    try {  
        Thread.sleep(500); // 500 milliseconds  
    } catch (InterruptedException e) { }  
}
```

- Thread.sleep() is a static method of class thread and can be invoked from any thread, including the main thread.

Pausing threads

```
for (int ctr=0; ctr < 500; ctr++) {  
    System.out.print(i);  
    try {  
        Thread.sleep(500); // 500 milliseconds  
    } catch (InterruptedException e) { }  
}
```

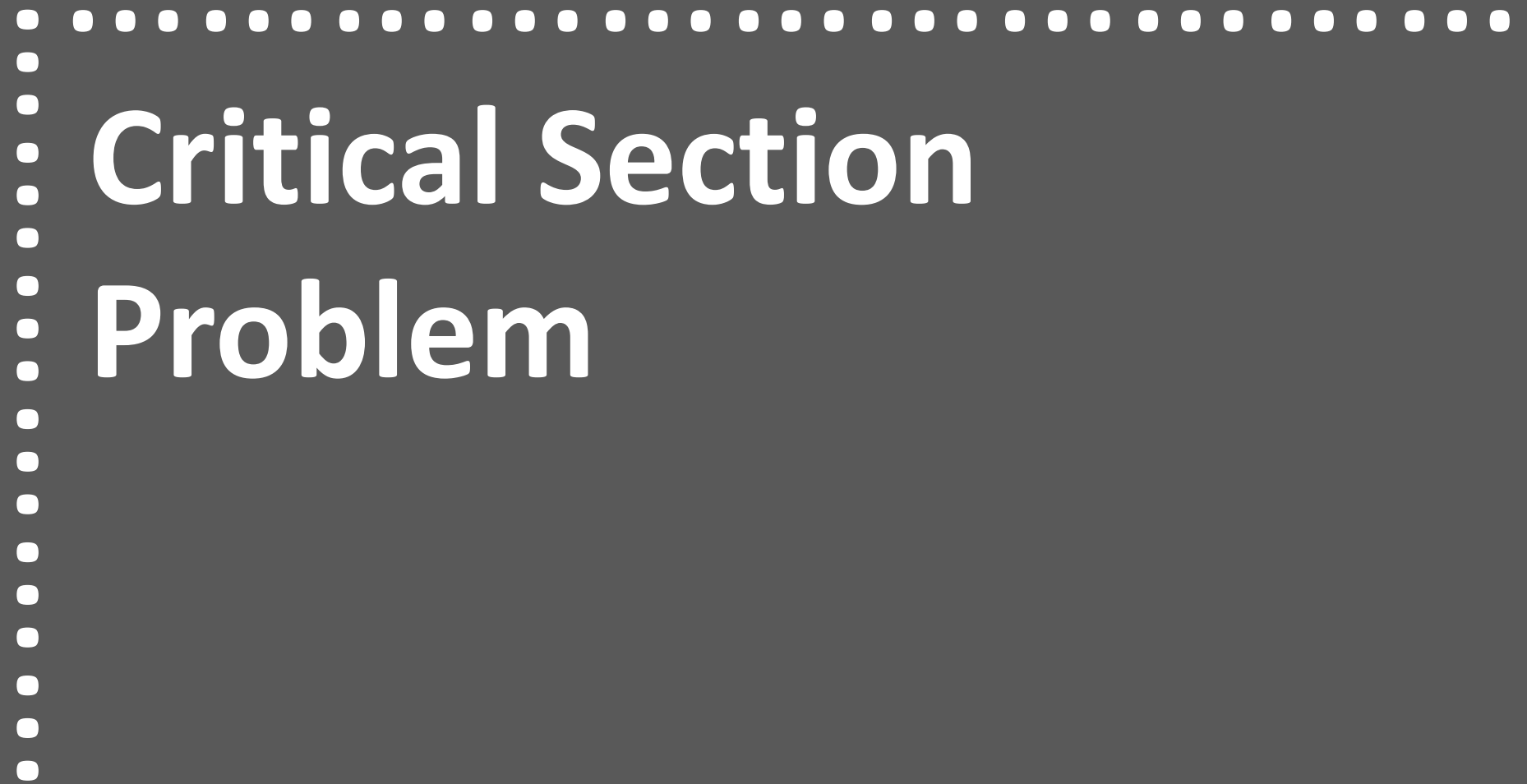
- The InterruptedException is an unchecked exception that is thrown by code that stops a thread from running.
 - Unchecked exception causing lines must be enclosed in a try-catch, in this case, Thread.sleep().
- An InterruptedException is thrown when a thread is waiting, sleeping, or otherwise paused for a long time and another thread interrupts it using the interrupt() method in class Thread.

Joins

- You can also make a thread stop until another thread finishes running by invoking a thread's `join()` method.
- For instance, to make our main thread to stop running until thread `t1` is finished, we could simply say...

```
public static void main(String args[]) {  
    ...  
    t1.start();  
    try {  
        t1.join();  
    } catch (InterruptedException e) { }  
    t2.start();  
}
```

- This would cause all the 1's to be printed out before thread 2 and thread 3 start.

A graphic consisting of a horizontal dotted line and a vertical dotted line that meet at a right angle, forming an L-shape. The horizontal line extends across the top of the text area, and the vertical line extends down the left side of the text area.

Critical Section Problem

The Critical Section Problem

- This section we will find out how to implement a solution to the critical section problem using Java
- Recall that only a single process can enter its critical section, all other processes would have to wait
 - No context switching occurs inside a critical section

The Critical Section Problem

- Instead of printing a continuous stream of numbers, MyThread calls a `print10()` method in a class `MyPrinter`
 - `print10()` prints 10 continuous numbers on a single line before a newline.
- Our goal is to have these 10 continuous numbers printed without any context switches occurring.
 - Our output should be:

```
...  
1111111111  
1111111111  
2222222222  
3333333333  
1111111111  
...
```


MyPrinter

- We define a class MyPrinter that will have our print10() method

```
class MyPrinter {  
    public void print10(int value) {  
        for (int i = 0; i < 10; i++) {  
            System.out.print(value);  
        }  
        System.out.println(""); // newline after 10 numbers  
    }  
}
```

MyThread (revised)

- Instead of printing numbers directly, we use the print10() method in MyThread, as shown by the following

```
class MyThread extends Thread {  
    int i;  
    MyPrinter p;  
    MyThread(int i) {  
        this.i = i;  
        p = new MyPrinter();  
    }  
    public void run() {  
        for (int ctr=0; ctr < 500; ctr++) {  
            p.print10(i);  
        }  
    }  
}
```

MyThreadDemo (revised)

- First, we will try to see the output of just a single thread running.

```
class MyThreadDemo {  
    public static void main(String args[]) {  
        MyThread t1 = new MyThread(1);  
        // MyThread t2 = new MyThread(2);  
        // MyThread t3 = new MyThread(3);  
        t1.start();  
        // t2.start();  
        // t3.start();  
        System.out.print("Main ends");  
    }  
}
```

*We comment out
the other threads
for now...*

MyThreadDemo

- When we run our MyThreadDemo, we can see that the output really does match what we want.

```
> java MyThreadDemo  
1111111111  
1111111111  
1111111111  
1111111111  
1111111111  
...
```

MyThreadDemo (revised 2)

- However, let us try to see the output with other threads.

```
class MyThreadDemo {  
    public static void main(String args[]) {  
        MyThread t1 = new MyThread(1);  
        MyThread t2 = new MyThread(2);  
        MyThread t3 = new MyThread(3);  
        t1.start();  
        t2.start();  
        t3.start();  
        System.out.print("Main ends");  
    }  
}
```

*We run all three
threads*

MyThreadDemo

- Our output would look like the following

```
> java MyThreadDemo  
1111111111  
111112222222  
1111  
22233333332  
...
```

- We do not achieve our goal of printing 10 consecutive numbers in a row
 - all three threads are executing the print10's method at the same time, thus having more than one number appearing on a single line.

Critical Section Problem

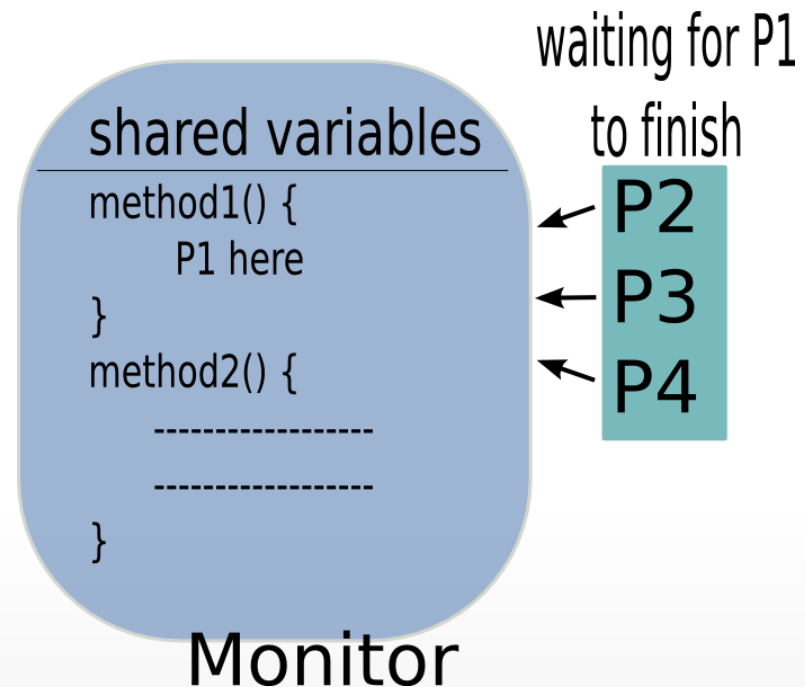
- To achieve our goal there should not be any context switches happening inside `print10()`
 - Only a single thread should be allowed to execute inside `print10()`
- As can be seen, this is an example of the critical section problem
- To solve this, we can use some of the techniques discussed in our synchronization chapter

Possible Solutions

- Busy Wait
- Wait and Notify
- Semaphores
- Monitors

Synchronized keyword

- Now we will discuss Java's solution to the critical section problem
- Java uses a monitor construct
 - Only a single thread may run inside the monitor



Even if P2 is calling method2(), it has to wait for P1 to finish

Synchronized keyword

- To turn an object into a monitor, simply put the synchronized keyword on your method definitions
 - Only a single thread can run any synchronized method in an object

```
class MyPrinter {  
    public synchronized void print10(int value) {  
        for (int i = 0; i < 10; i++) {  
            System.out.print(value);  
        }  
        System.out.println(""); // newline after 10  
        numbers  
    }  
}
```

Still not working!

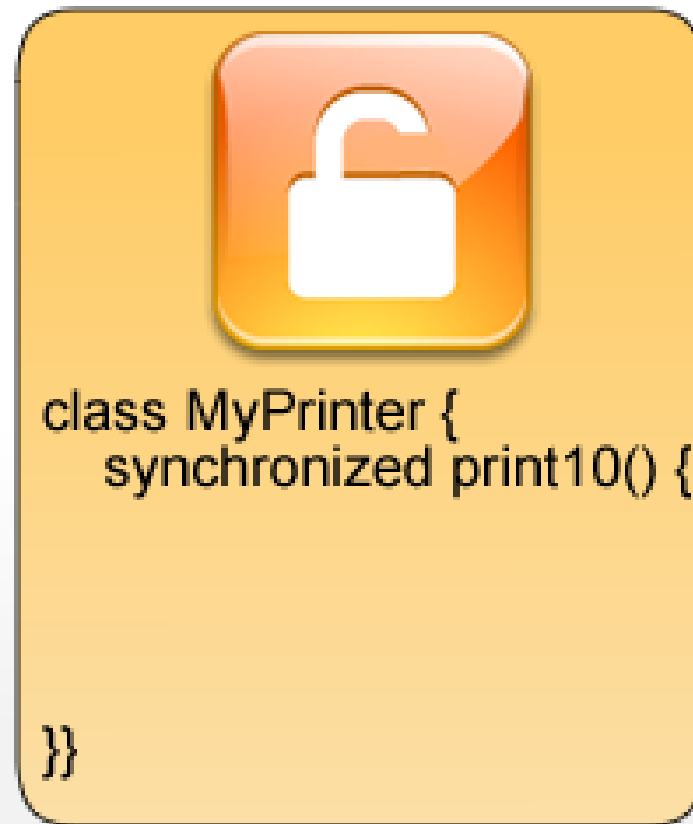
- However, even if we did turn our print10() method into a synchronized method, it will still not work

```
> java MyThreadDemo
1111111111
111112222222
1111
22233333332
...
```

- To find out how to make it work, we need to first find out how the synchronized keyword works

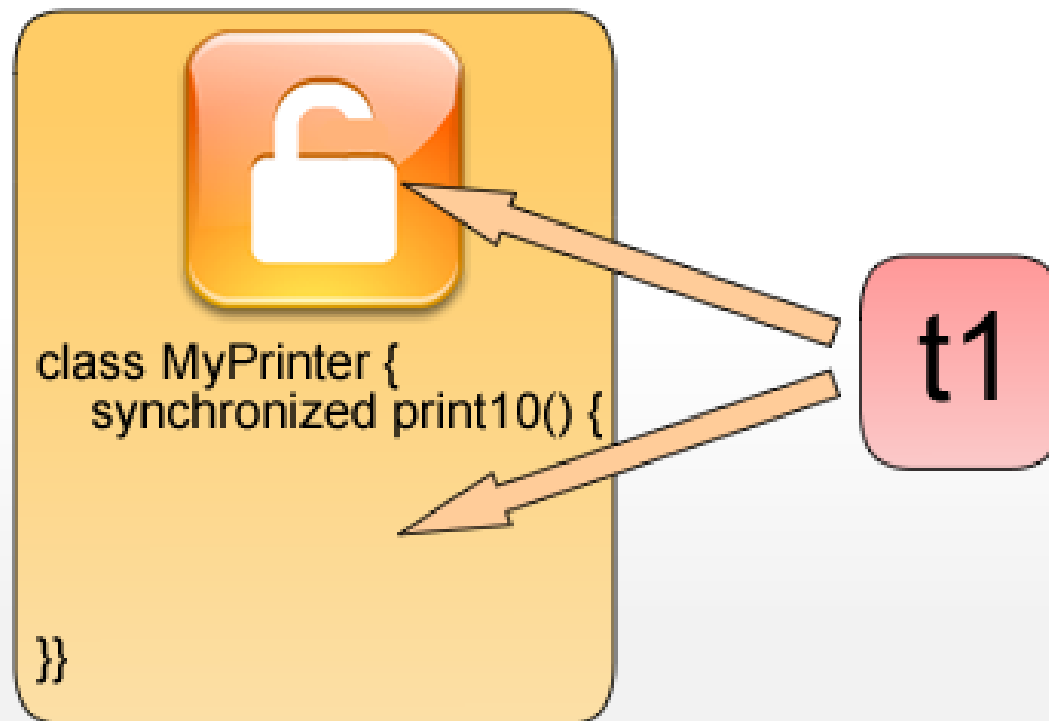
Intrinsic locks

- Every object in Java has an intrinsic lock



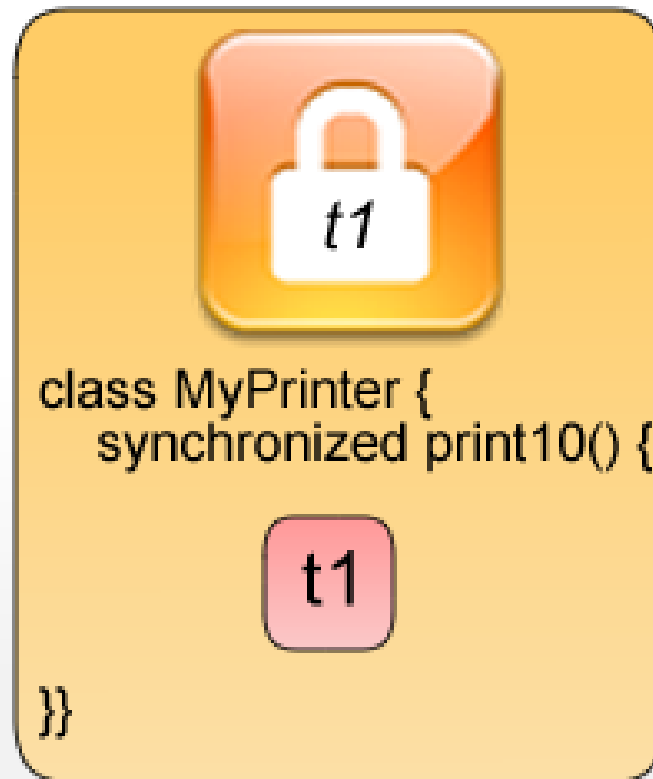
Intrinsic locks

- When a thread tries to run a synchronized method, it first tries to get a lock on the object



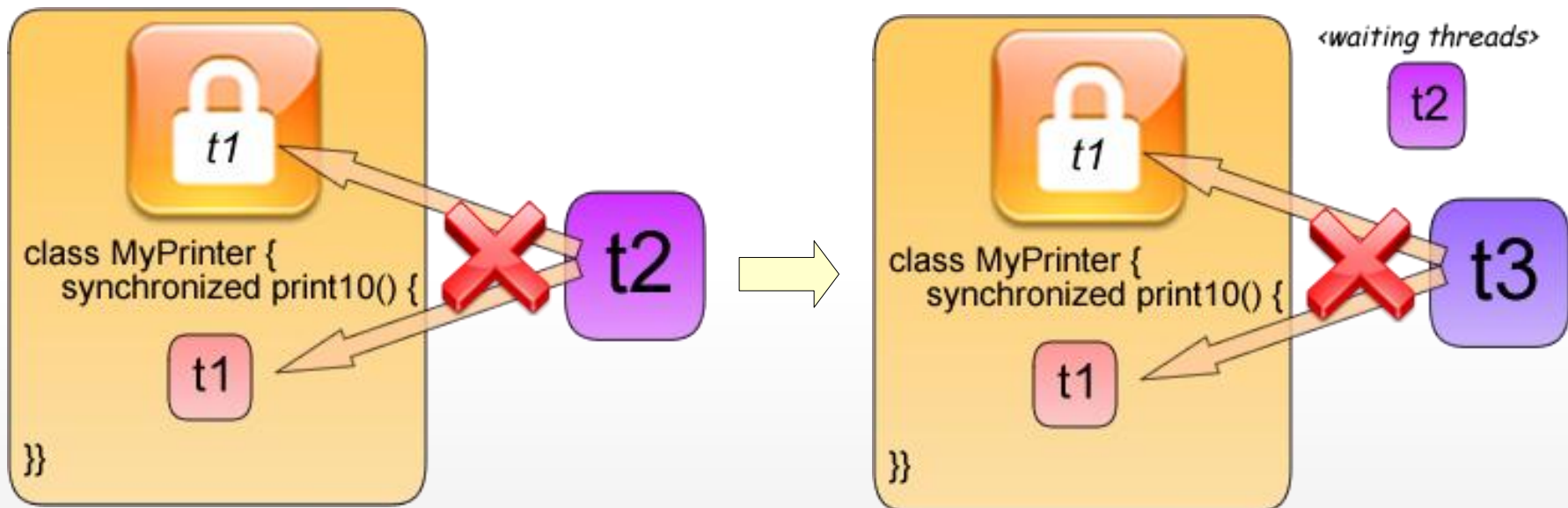
Intrinsic locks

- If a thread is successful, then it owns the lock and executes the synchronized code.



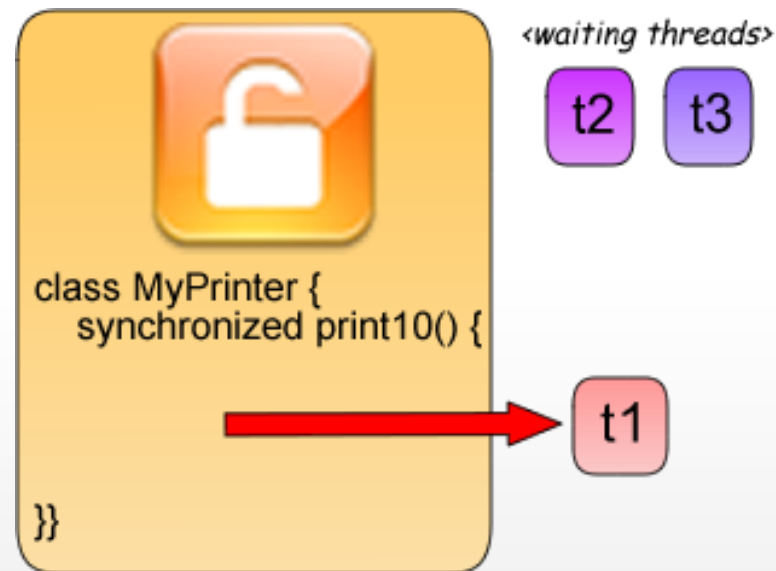
Intrinsic locks

- Other threads cannot run the synchronized code because the lock is unavailable



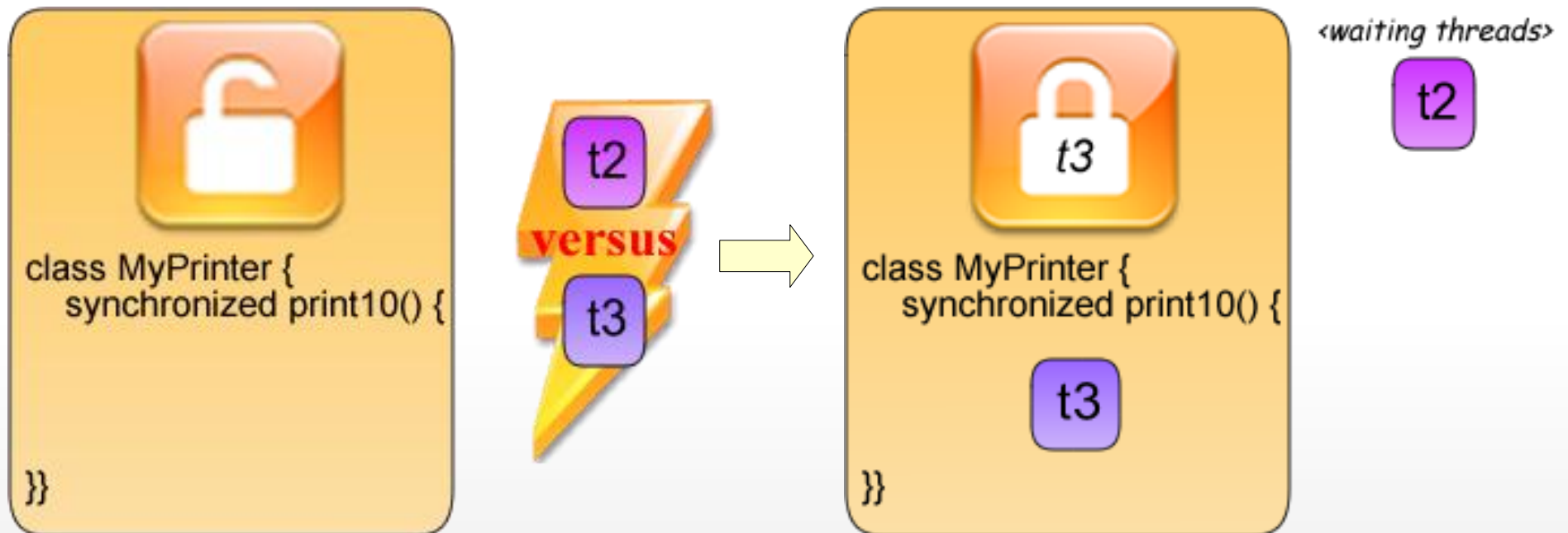
Intrinsic Locks

- Threads can only enter once the thread owning the lock leaves the synchronized method



Intrinsic Locks

- Waiting threads would then compete on who gets to go next

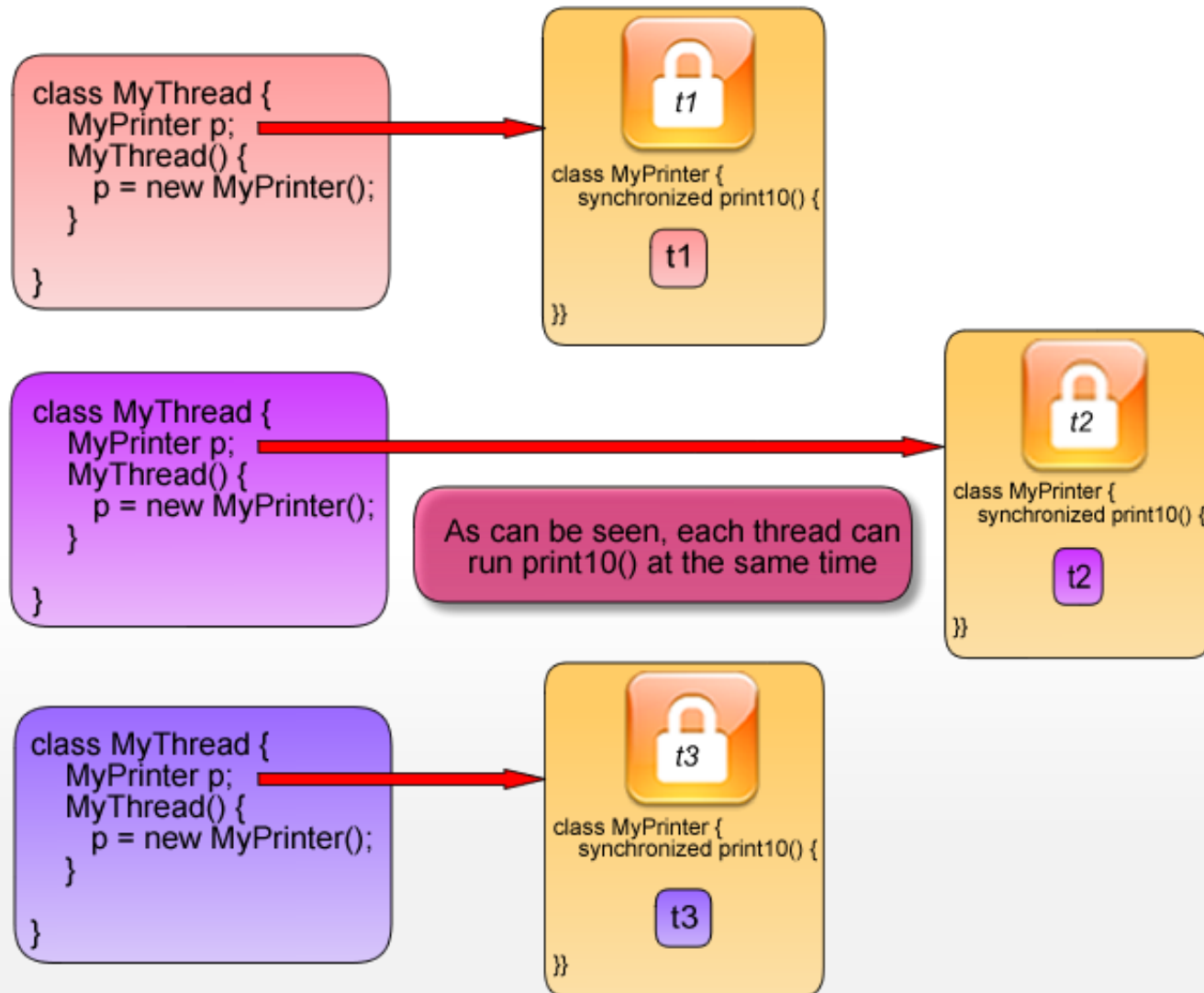


MyThreadDemo Failure

- So why doesn't our example work?
- Because each thread has its own copy of the MyPrinter object!

```
class MyThread extends Thread {  
    int i;  
    MyPrinter p;  
    MyThread(int i) {  
        this.i = i;  
        p = new MyPrinter();    // each MyThread creates  
its own MyPrinter!  
    }  
    public void run() {  
        for (int ctr=0; ctr < 500; ctr++) {  
            p.print10(i);  
        }  
    }  
}
```

Different Locks



Important

- Recall our definition of synchronized methods
 - Only a single thread can run any synchronized method in an object
- Since each thread has its own MyPrinter object, then no locking occurs

Solution

- So, to make it work, all threads must point to the same MyPrinter object

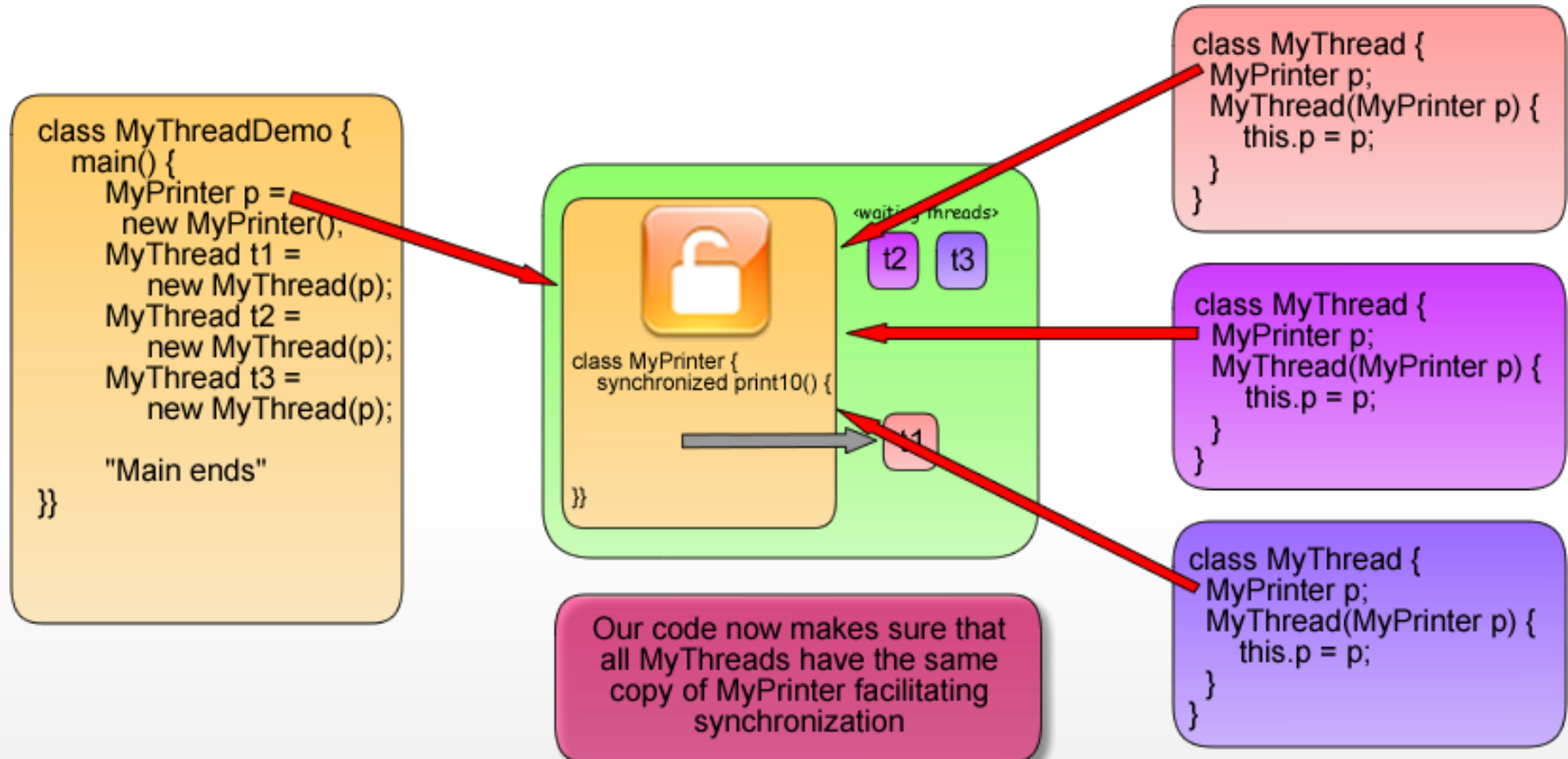
Solution

- Note how all MyThreads now have the same MyPrinter object

```
class MyThread extends Thread {  
    int i;  
    MyPrinter p;  
    MyThread(int i, MyPrinter p) {  
        this.i = i; this.p = p  
    }  
    public synchronized void run() {  
        for (int ctr=0; ctr < 500;  
ctr++) {  
                                p.print10(i);  
        }  
    }  
}
```

```
class MyThreadDemo {  
    public static void main(String args[]) {  
        MyPrinter p = new  
        MyPrinter();  
        MyThread t1 = new  
        MyThread(1,p);  
        MyThread t2 = new  
        MyThread(2,p);  
        MyThread t3 = new  
        MyThread(3,p);  
        t1.start();  
        t2.start();  
        t3.start();  
    }  
}
```

Solution



Locks and Doors

- A way to visualize it is that all Java objects can be doors
 - A thread tries to see if the door is open
 - Once the thread goes through the door, it locks it behind it,
 - No other threads can enter the door because our first thread has locked it from the inside
 - Other threads can enter if the thread inside unlocks the door and goes out
 - Lining up will only occur if everyone only has a single door to the critical region

Synchronized methods

- Running MyThreadDemo now correctly shows synchronized methods at work

```
>java MyThreadDemo  
1111111111  
1111111111  
1111111111  
2222222222  
3333333333  
...
```

Remember

- Only a single thread can run any synchronized method in an object
 - If an object has a synchronized method and a regular method, only one thread can run the synchronized method while multiple threads can run the regular method
- Threads that you want to have synchronized must share the same monitor object

Synchronized blocks

- Aside from synchronized methods, Java also allows for synchronized blocks.
- You must specify what object the intrinsic lock comes from

Synchronized blocks

- Our print10() implementation, done using synchronized statements, would look like this

```
class MyPrinter {  
    public void print10(int value) {  
        synchronized(this) {  
            for (int i = 0; i < 10; i++) {  
                System.out.print(value);  
            }  
            System.out.println(""); // newline after  
10 numbers  
        }  
    }  
}
```

We use a MyPrinter object for the lock, replicating what the synchronized method does

Synchronized blocks

- Synchronized blocks allow for flexibility, in the case if we want our intrinsic lock to come from an object other than the current object.
- For example, we could define MyThread's run method to perform the lock on the MyPrinter object before calling print10

```
class MyThread extends Thread {  
    MyPrinter p;  
    ...  
    public void run() {  
        for (int i = 0; i < 100; i++) {  
            synchronized(p) {  
                p.print10(value);  
            }  
        }  
    }  
}
```

*Note that any lines
before and after our
synchronized block can
be executed
concurrently by threads*

Multiple locks

- Also, the use of the synchronized block allows for more flexibility by allowing different parts of code to be locked with different objects.

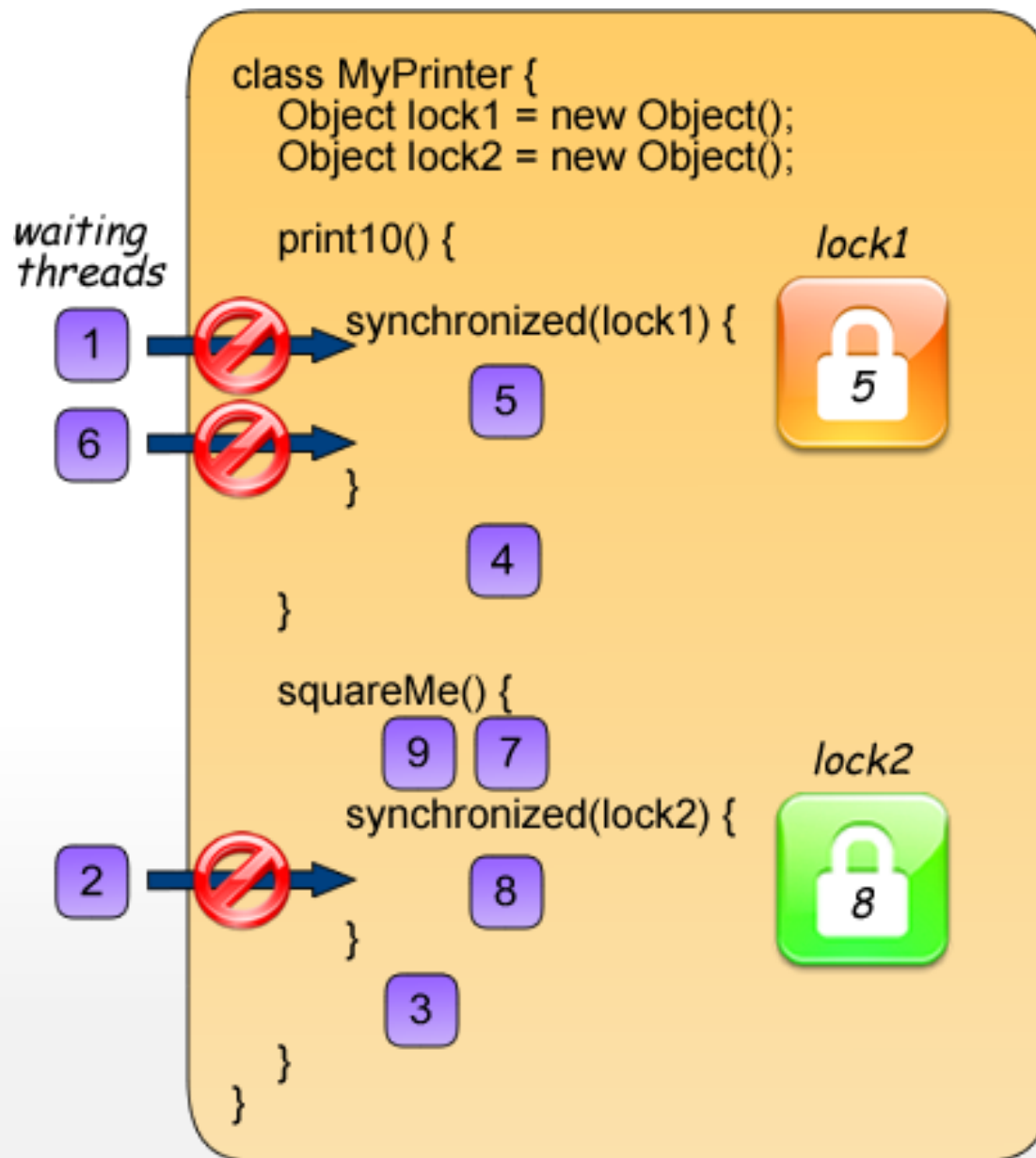
Multiple locks

- For example, consider a modified MyPrinter which has two methods, which we'll allow to run concurrently

```
class MyPrinter {
    Object lock1 = new Object();
    Object lock2 = new Object();
    public void print10(int value) {
        synchronized(lock1) {
            for (int i = 0; i < 10; i++) {
                System.out.print(value);
            }
            System.out.println(""); // newline after 10 numbers
        }
    }
    public int squareMe(int i) {
        synchronized (lock2) {
            return i * i;
        }
    }
}
```

The blocks inside print10() and squareMe() can run at the same time because they use different objects for their locking mechanism

Synchronization still applies. For example only a single thread can run the synchronized block in squareMe() at any point in time



Any object in Java can be used as a lock

Thread 5 and 8 are allowed at the same time in synchblock as they use different locks

Only one thread in synchronized block. As many threads everywhere else

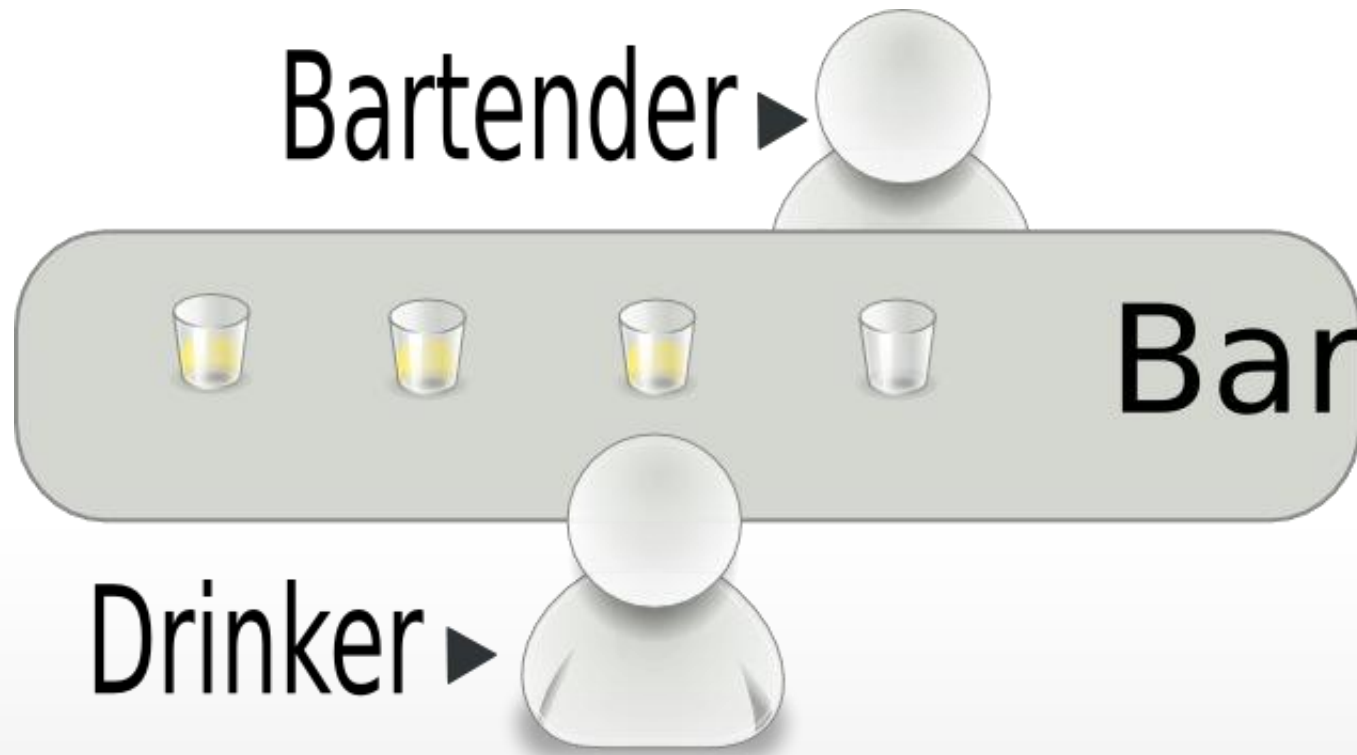
Outline

- Creating a Java Thread
- Synchronized Keyword
- **Wait and Notify**
- High Level Cuncurrency Objects

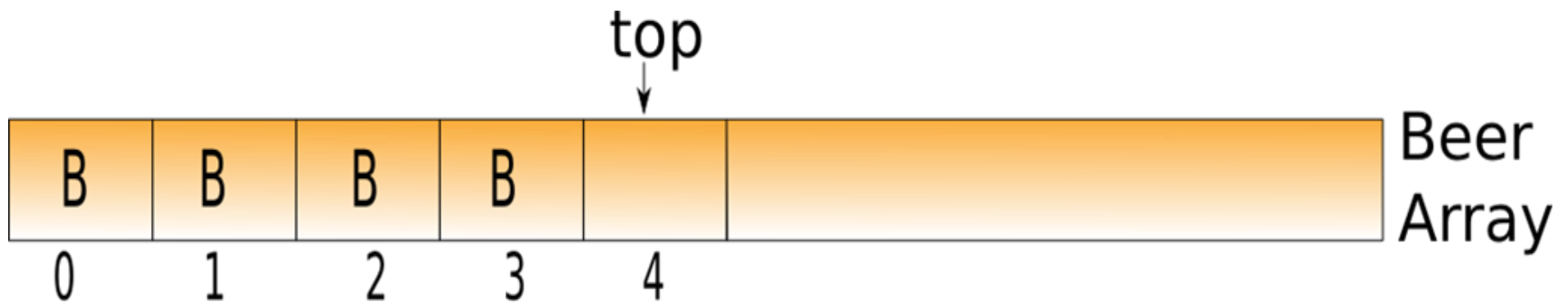
Producer-Consumer problem

- We will now discuss a solution to the Producer-Consumer problem.
- Instead of using beer, the producer will store in a shared stack a random integer value which will be retrieved by the consumer process.

Producer-Consumer



Producer-Consumer Code



BARTENDER

```
beerstack[top] = new Beer()
```

```
top = top+1
```

BEERDRINKER

```
top = top - 1
```

```
drink(beerstack[top])
```

Producer-Consumer

- As can be seen, we would need to have a place to store our array of integers and our top variable, which should be shared among our producer and consumer.
 - MAXCOUNT is the maximum number of items the Producer produces
 - We'll assume the maximum array size of 10000 for now

```
class SharedVars {  
    final int MAXCOUNT = 100;  
    int array[] = new int[10000];  
    int top;  
  
    // <more code to follow>  
}
```

Producer-Consumer

- To keep things object-oriented, we will place the code for placing values in the stack and getting values in the stack in our SharedVars class
- Our next slide shows our implementation

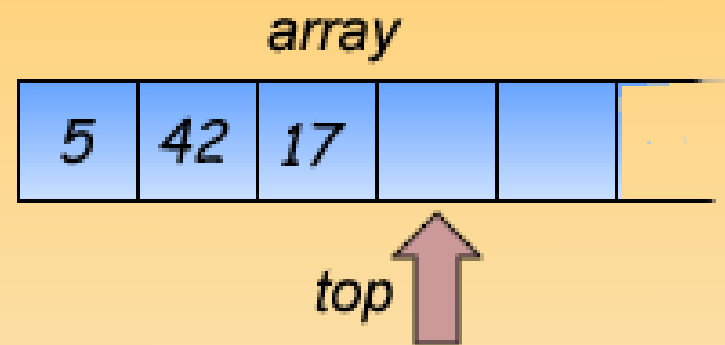
SharedVars

```
class SharedVars {
    final int MAXCOUNT = 100;
    int array[] = new int[10000];
    int top;

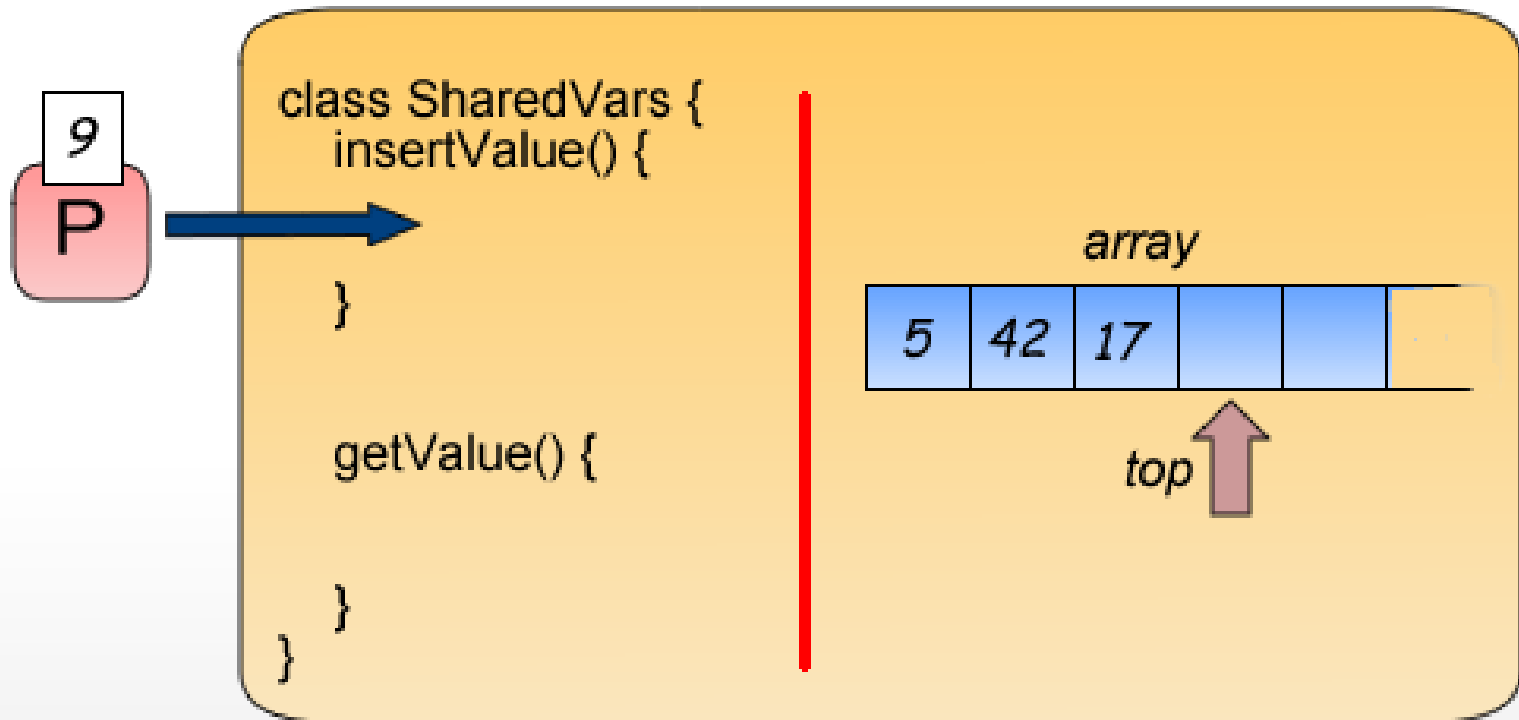
    // method for inserting a value
    public void insertValue(int value) {
        if (top < array.length) {
            array[top] = value;
            top++;
        }
    }
    // method for getting a value
    public int getValue() {
        if (top > 0) {
            top--;
            return array[top];
        }
        else
            return -1;
    }
}
```

SharedVars

```
class SharedVars {  
    insertValue() {  
  
    }  
  
    getValue() {  
  
    }  
}
```



SharedVars insertValue()



SharedVars insertValue()

```
class SharedVars {  
    insertValue() {
```

P

```
}
```

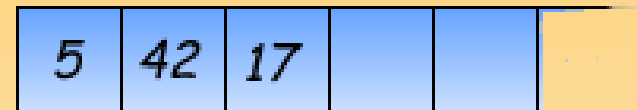
```
    getValue() {
```

```
}
```

```
}
```

9

array



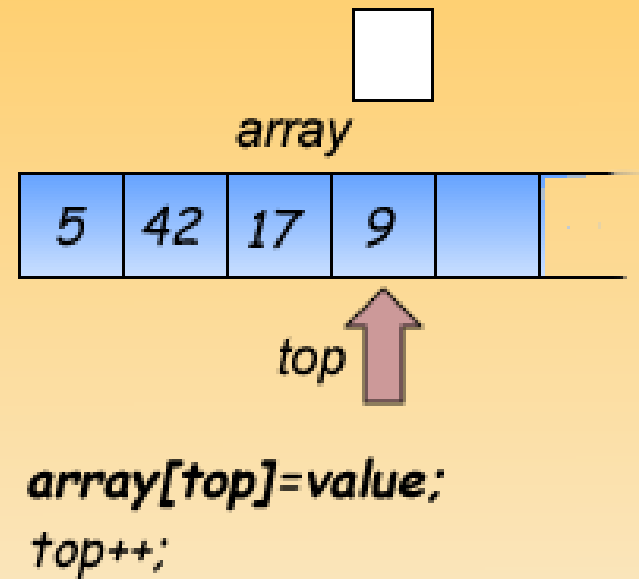
top



```
array[top]=value;  
top++;
```

SharedVars insertValue()

```
class SharedVars {  
    insertValue() {  
        P  
    }  
  
    getValue() {  
    }  
}
```



SharedVars insertValue()

```
class SharedVars {  
    insertValue() {
```

P

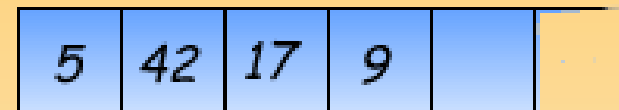
```
}
```

```
    getValue() {
```

```
}
```

```
}
```

array



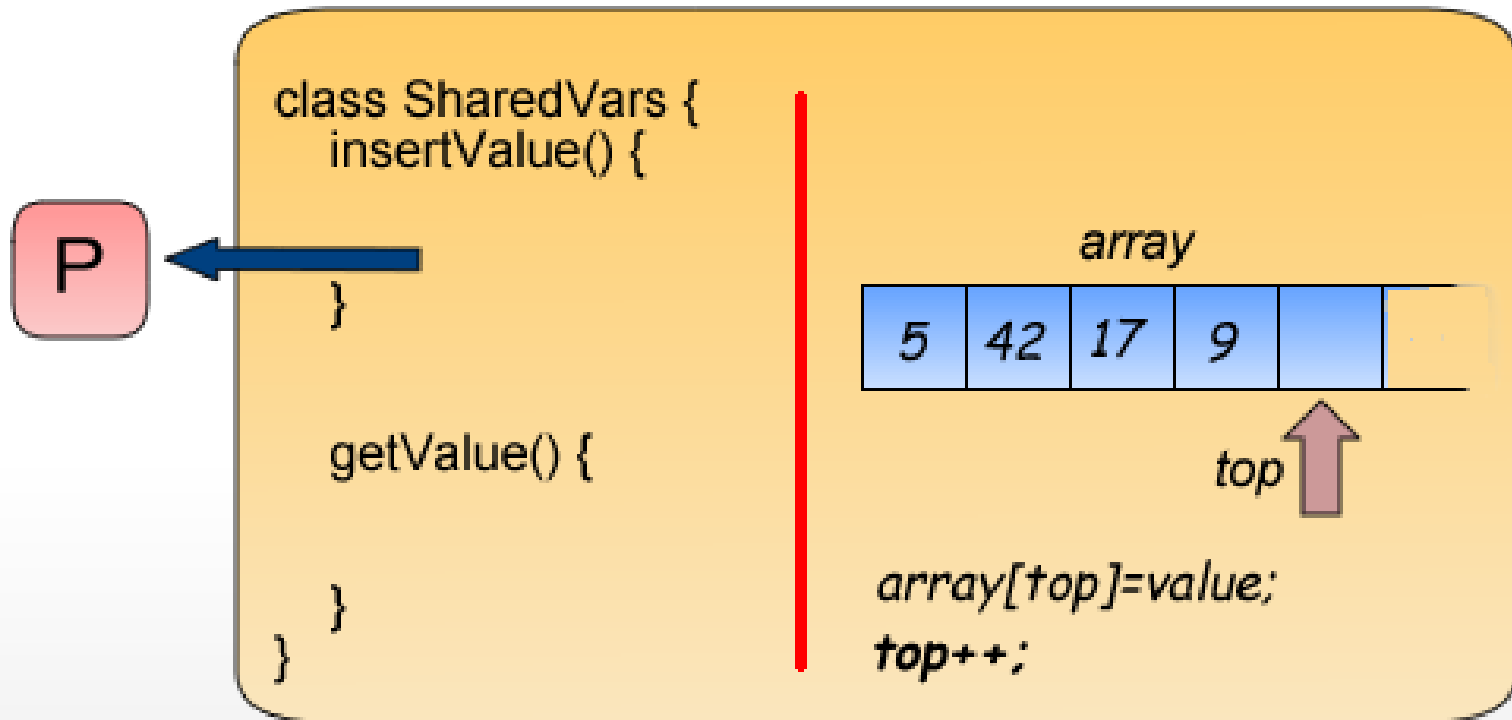
top



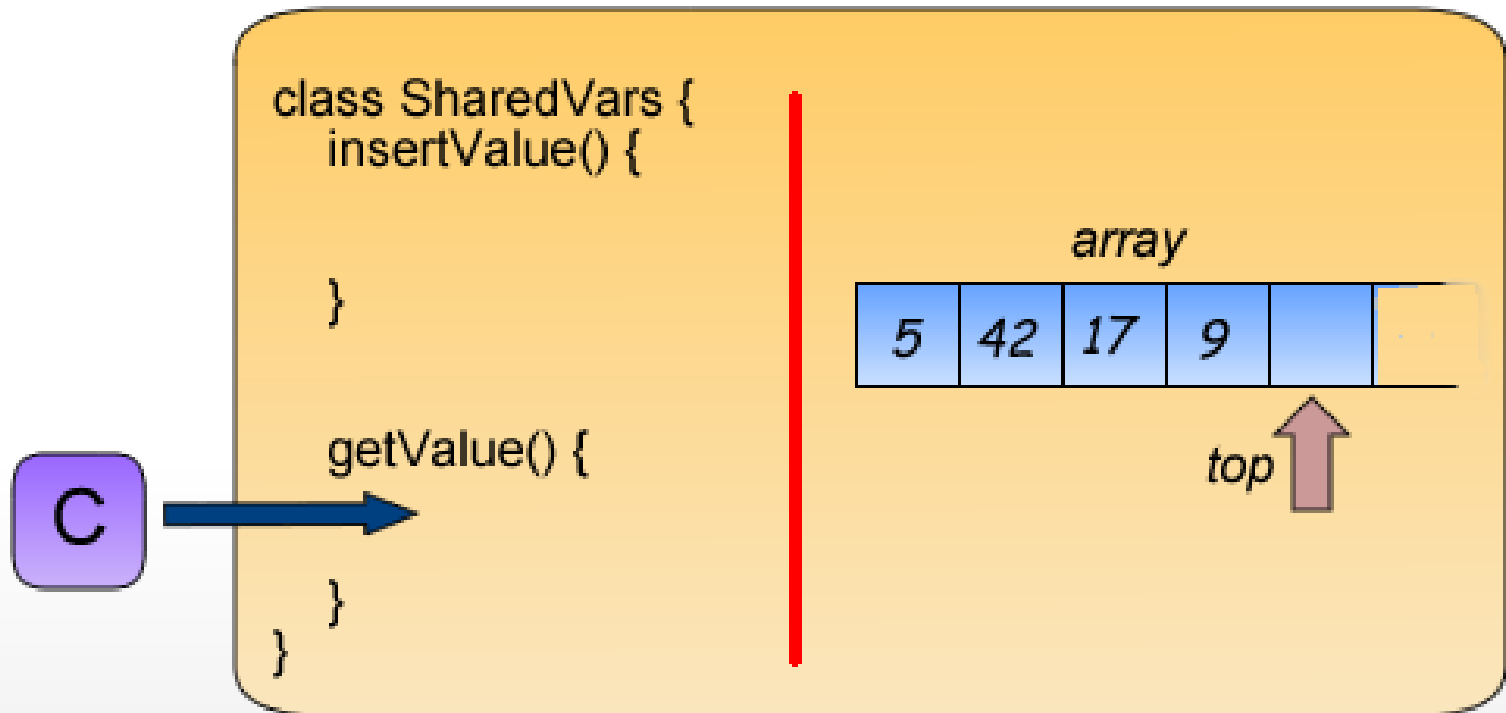
```
    array[top]=value;
```

```
    top++;
```

SharedVars insertValue()

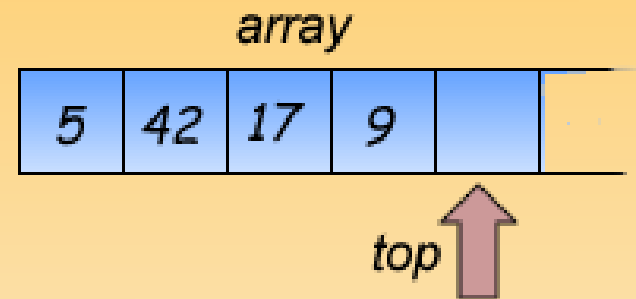


SharedVars getValue()



SharedVars getValue()

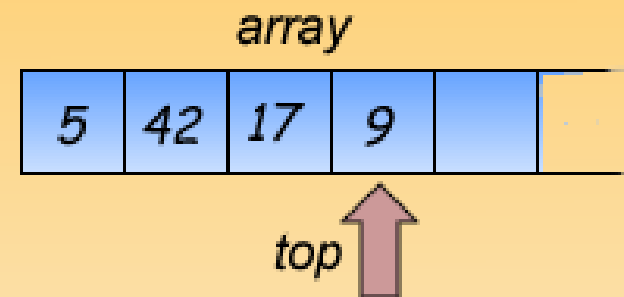
```
class SharedVars {  
    insertValue() {  
  
    }  
  
    getValue() {  
        C  
    }  
}
```



```
top--;  
return array[top];
```

SharedVars getValue()

```
class SharedVars {  
    insertValue() {  
  
    }  
  
    getValue() {  
        C  
    }  
}
```



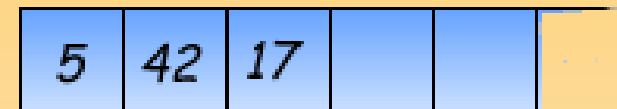
```
top--;  
return array[top];
```


SharedVars getValue()

```
class SharedVars {  
    insertValue() {  
  
    }  
  
    getValue() {  
        C  
    }  
}
```

Note: we don't really delete the value but we do so for simplicity's sake

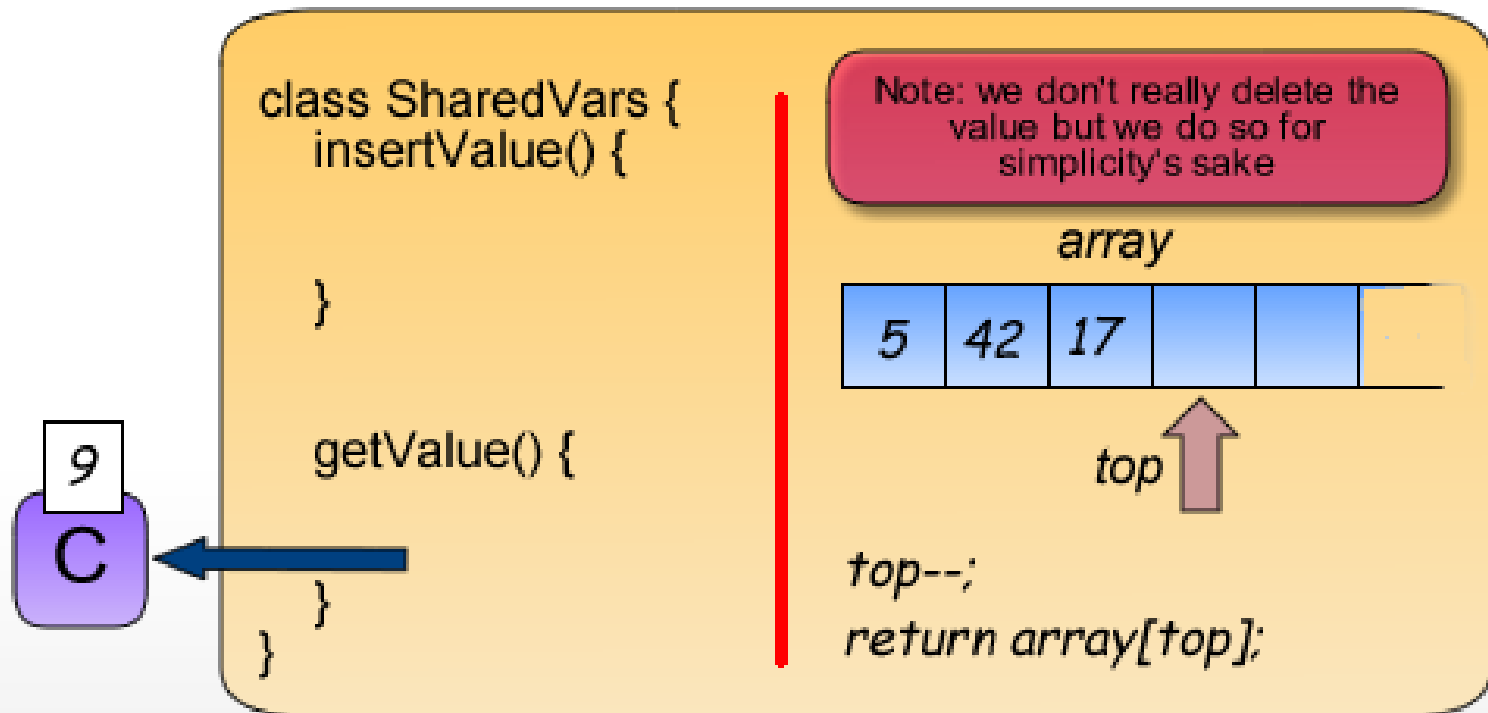
array



top ↑

top--; 9
return array[top];

SharedVars getValue()



Producer-Consumer

- Now we will slowly build our Producer and Consumer class
- We can say that our Producer and Consumer should be implemented as threads.
 - Thus we have our code below, including a class that starts these threads

```
class Producer extends Thread {  
    public void run() {  
    }  
}
```

```
class Consumer extends Thread {  
    public void run() {  
    }  
}
```

```
class PCDemo {  
    public static void main(String args[]) {  
        Producer p = new Producer();  
        Consumer c = new Consumer();  
        p.start();  
        c.start();  
    }  
}
```

Producer-Consumer

- Producer and Consumer should have the same SharedVars object

```
class Producer extends Thread {  
    SharedVars sv;  
    Producer(SharedVars sv) {  
        this.sv = sv;  
    }  
    public void run() {  
    }  
}
```

```
class Consumer extends Thread {  
    SharedVars sv;  
    Consumer(SharedVars sv) {  
        this.sv = sv;  
    }  
    public void run() {  
    }  
}
```

```
class PCDemo {  
    public static void main(String args[]) {  
        SharedVars sv = new  
        SharedVars();  
        Producer p = new Producer(sv);  
        Consumer c = new Consumer(sv);  
        p.start();  
        c.start();  
    }  
}
```

Producer-Consumer

- There is no more need to modify our PCDemo class so we will not show it anymore.

Producer

- We want our producer to produce a random integer and store it in SharedVars. Thus our Producer code would look like this

```
class Producer extends Thread {  
    SharedVars sv;  
    Producer(SharedVars sv) {  
        this.sv = sv;  
    }  
    public void run() {  
        int value = (int)(Math.random() * 100); // random value from  
0 to 100  
        sv.insertValue(value);  
        System.out.println("Producer: Placing value: " + value);  
    }  
}
```

Producer

- We want our Producer to do this a hundred times, so we will add a loop to our code

```
class Producer extends Thread {
    SharedVars sv;
    Producer(SharedVars sv) {
        this.sv = sv;
    }
    public void run() {
        for (int i = 0; i < 100; i++) {
            int value = (int)(Math.random() * 100); // random
            value from 0 to 100
            sv.insertValue(value);
            System.out.println("Producer: Placing value: " +
            value);
        }
    }
}
```

Producer

- Finally, we have our producer pause for a maximum of 5 seconds each time it places a value

```
class Producer extends Thread {
    SharedVars sv;
    Producer(SharedVars sv) {
        this.sv = sv;
    }
    public void run() {
        for (int i = 0; i < 100; i++) {
            int value = (int)(Math.random() * 100); // random
value from 0 to 100

            sv.insertValue(value);
            System.out.println("Producer: Placing value: " +
value);

            try {
                Thread.sleep((int)(Math.random() *
5000)); // sleep 5s max
            } catch (InterruptedException e) { }
        }
    }
}
```


Consumer

- Now, we want our consumer to get a value from SharedVars

```
class Consumer extends Thread {  
    SharedVars sv;  
    Consumer(SharedVars sv) {  
        this.sv = sv;  
    }  
    public void run() {  
        int value = sv.getValue();  
        System.out.println("Consumer: I got value:" + value);  
    }  
}
```

Consumer

- We want to get a value 100 times

```
class Consumer extends Thread {  
    SharedVars sv;  
    Consumer(SharedVars sv) {  
        this.sv = sv;  
    }  
    public void run() {  
        for (int i = 0; i < 100; i++) {  
            int value = sv.getValue();  
            System.out.println("Consumer: I got value:" +  
value);  
        }  
    }  
}
```

Consumer

- Finally, just like producer, we pause for a maximum of 5 seconds on each loop

```
class Consumer extends Thread {
    SharedVars sv;
    Consumer(SharedVars sv) {
        this.sv = sv;
    }
    public void run() {
        for (int i = 0; i < 100; i++) {
            int value = sv.getValue();
            System.out.println("Consumer: I got value:" +
value);

            try {
                Thread.sleep((int)(Math.random() *
5000)); // sleep 5s max
            } catch (InterruptedException e) { }
        }
    }
}
```

Race Condition

- As was discussed in a previous chapter, we try to avoid a race condition between insertValue() and getValue()

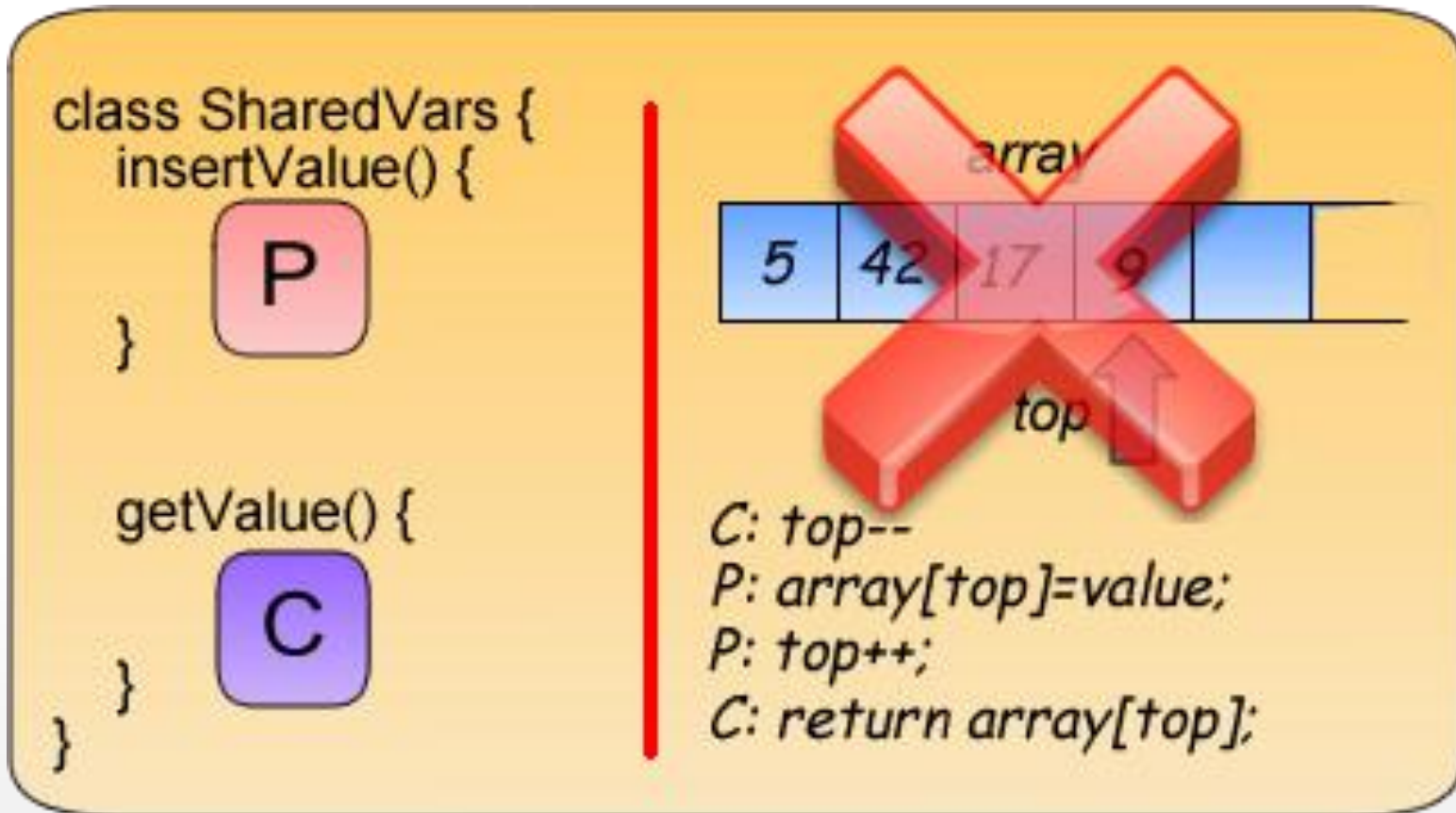
Consumer getValue(): $top = top - 1$

Producer insertValue(): $array[top] = value$ // this would overwrite an existing value!

Producer insertValue(): $top = top + 1$

Consumer getValue(): $return array[top]$ // consumer returns an already returned value

Race Condition



Race Condition

- We do not want `getValue()` and `insertValue()` to run at the same time
- Therefore, we modify `SharedVars` to use synchronized blocks
 - We could also use synchronized methods but we need some non-synchronized commands

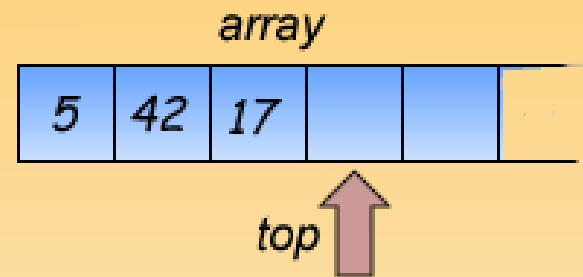
Synchronized SharedVars

```
class SharedVars {  
    final int MAXCOUNT = 100;  
    int array[] = new int[10000];  
    int top;  
  
    public void insertValue(int value) { // method for inserting a value  
        synchronized(this) {  
            if (top < array.length) {  
                array[top] = value;  
                top++;  
            }  
        }  
    }  
  
    public int getValue() { // method for getting a value  
        synchronized(this) {  
            if (top > 0) {  
                top--;  
                return array[top];  
            }  
            else  
                return -1;  
        }  
    }  
}
```

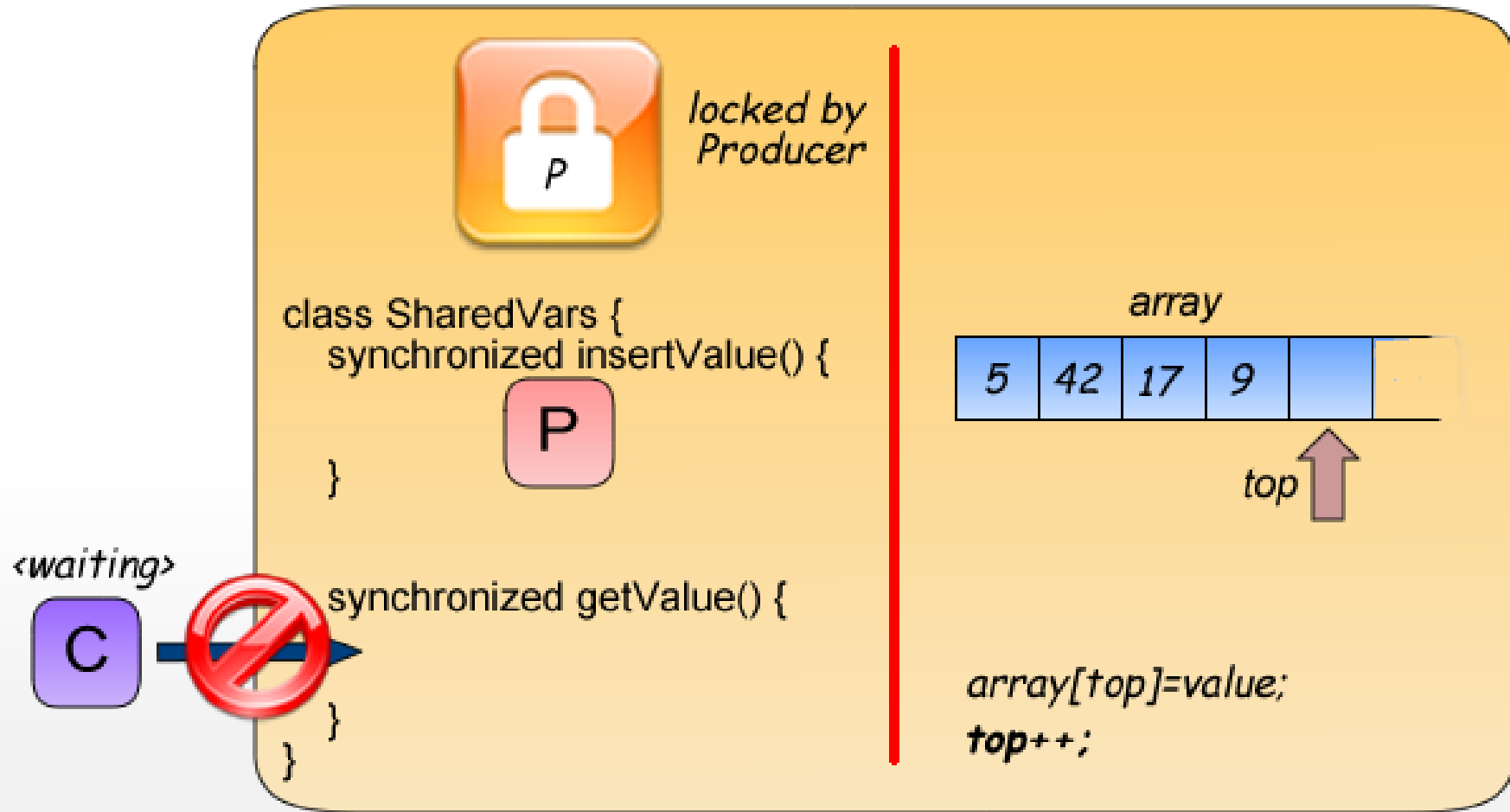
Synchronized SharedVars



```
class SharedVars {  
    synchronized insertValue() {  
  
    }  
  
    synchronized getValue() {  
  
    }  
}
```



Synchronized SharedVars



Producer-Consumer

- Upon executing our program, the output of our code would look something like this:
 - Producer inserts value: 15
 - Consumer got: 15
 - Consumer got: -1
 - Producer inserts value: 50
 - Producer inserts value: 75
 - Consumer got: 75
 - ...
- Note that both Consumer and Producer are running at the same time.

Got -1?

Producer inserts value: 15

Consumer got: 15

Consumer got: -1

Producer inserts value: 50

Producer inserts value: 75

Consumer got: 75

...

- Notice that, if we try to get a value from the array and there isn't any, we return a value -1.
 - You can see this clearly if you increase the Producer delay to 10s.
 - Producer adds a value every 10s, consumer gets one every 5s
- Wouldn't it be better if we have our Consumer wait for the Producer to produce something instead of just getting -1?

Busy wait

- We could implement a busy wait for this

```
public int getValue() {  
    while (top <= 0) { } // do nothing  
    synchronized(this) {  
        top--;  
        return array[top];  
    }  
}
```

- Note how we placed this outside our synchronized block
 - Having our busy wait inside would mean blocking out producer thread from insertValue(), which is what is needed to break the busy wait

The wait() method

- A better solution is to use the wait() method, defined in class Object, and is therefore inherited by all objects
- A thread invoking wait() will suspend the thread
- However, a thread invoking wait() must own the intrinsic lock of the object it is calling wait() from
 - If we are going to call this.wait() it has to be in synchronized(this) block
- Also, as with all methods that suspend thread execution, like join() and sleep(), our wait() method must be in a try-catch block that catches InterruptedExceptions.

The wait() method

```
// called by Consumer thread
public int getValue() {
    synchronized(this) {
        if (top <= 0) {
            try {
                this.wait();
            } catch (InterruptedException e) { }
        }
        top--;
        return array[top];
    }
}
```

The wait() method

- All threads that call wait() on an object are placed in a pool of waiting threads for that object.
- Execution resumes when another thread calls the notify() method of the object our first thread is waiting on
 - The notify() method is defined in class Object.
 - All objects have wait() and notify()

Wait() and notify()

- For our example, our producer thread, after inserting on an empty array, would notify the consumer thread that it has placed a value in our array by calling the notify() method on the SharedVars object
 - Recall that Producer and Consumer both have a reference to the same SharedVars object.
 - The producer calling notify() from insertValue() would inform any the consumer waiting on the same SharedVar object.

The notify() method

```
// called by producer
public void insertValue(int value) {
    synchronized(this) {
        if (top < array.length) {
            array[top] = value;
            top++;
            if (top == 1) { // we just inserted on an empty array
                this.notify(); // notify sleeping thread
            }
        }
    }
}
```

Final SharedVars

```
class SharedVars {
    final int MAXCOUNT = 100;
    int array[] = new int[10000];
    int top;

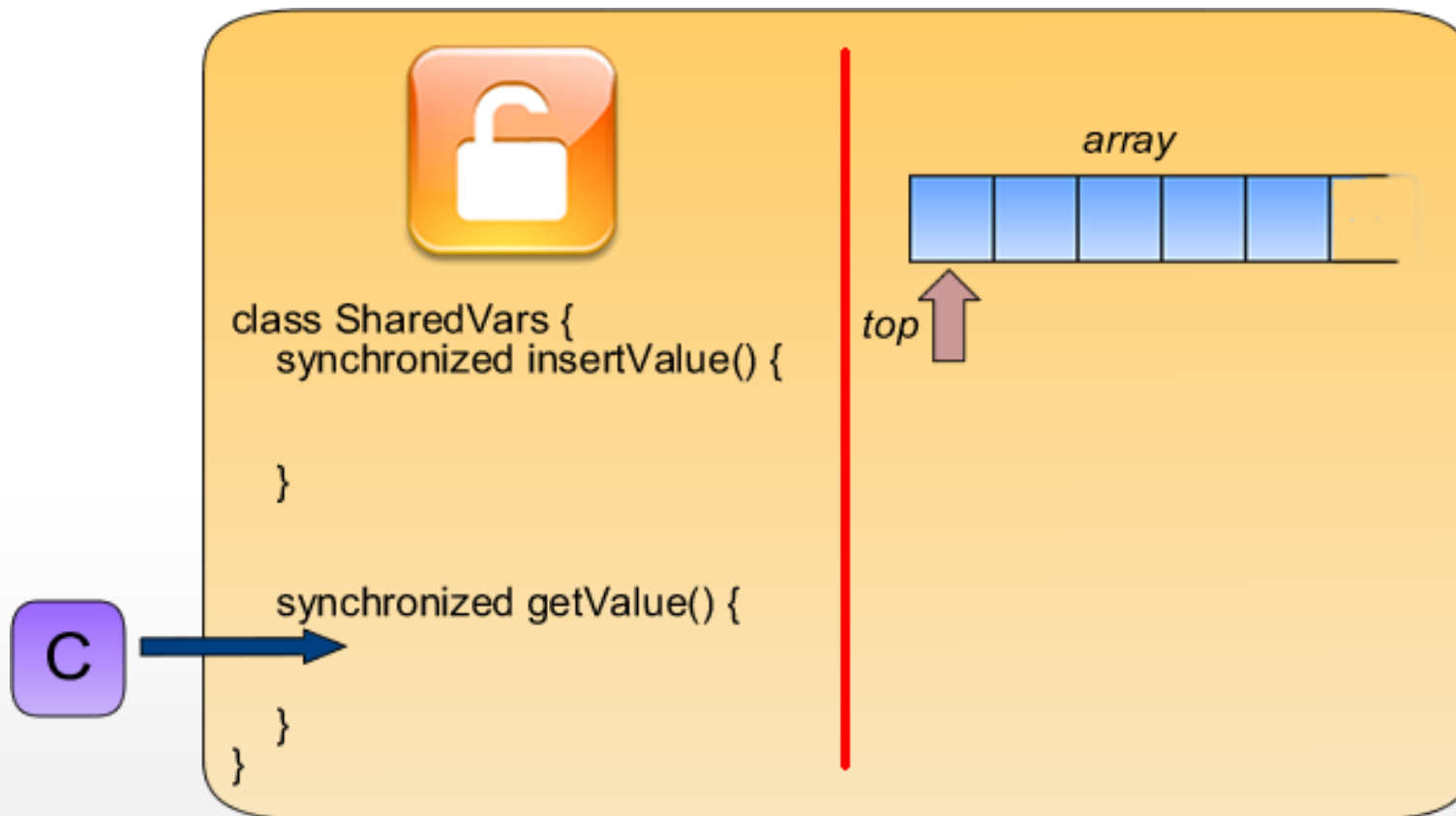
    public void insertValue(int value) {
        synchronized(this) {
            if (top < array.length) {
                array[top] = value;
                top++;
                if (top == 1) { // we just inserted on an empty array
                    this.notify(); // notify any sleeping thread
                }
            }
        }
    }

    public int getValue() {
        synchronized(this) {
            if (top <= 0) {
                try {
                    this.wait();
                } catch (InterruptedException e) { }
            }
            top--;
            return array[top];
        }
    }
}
```

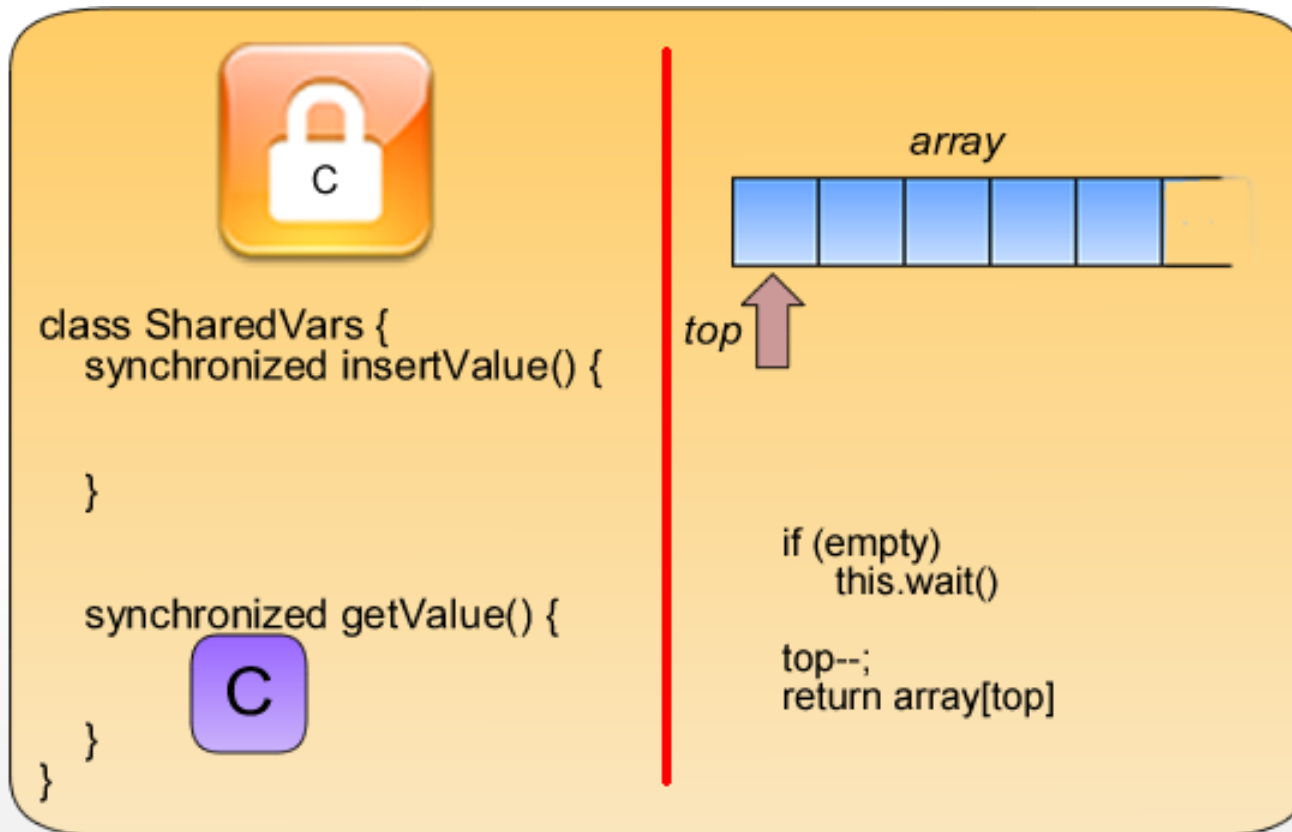
Wait() and notify()

- When the notify() method of an object is called, then a single waiting thread on that object is signaled to get ready to resume execution.
- After our Producer thread exits insertValue and releases the lock, our Consumer thread gets the lock once again and resumes its execution.

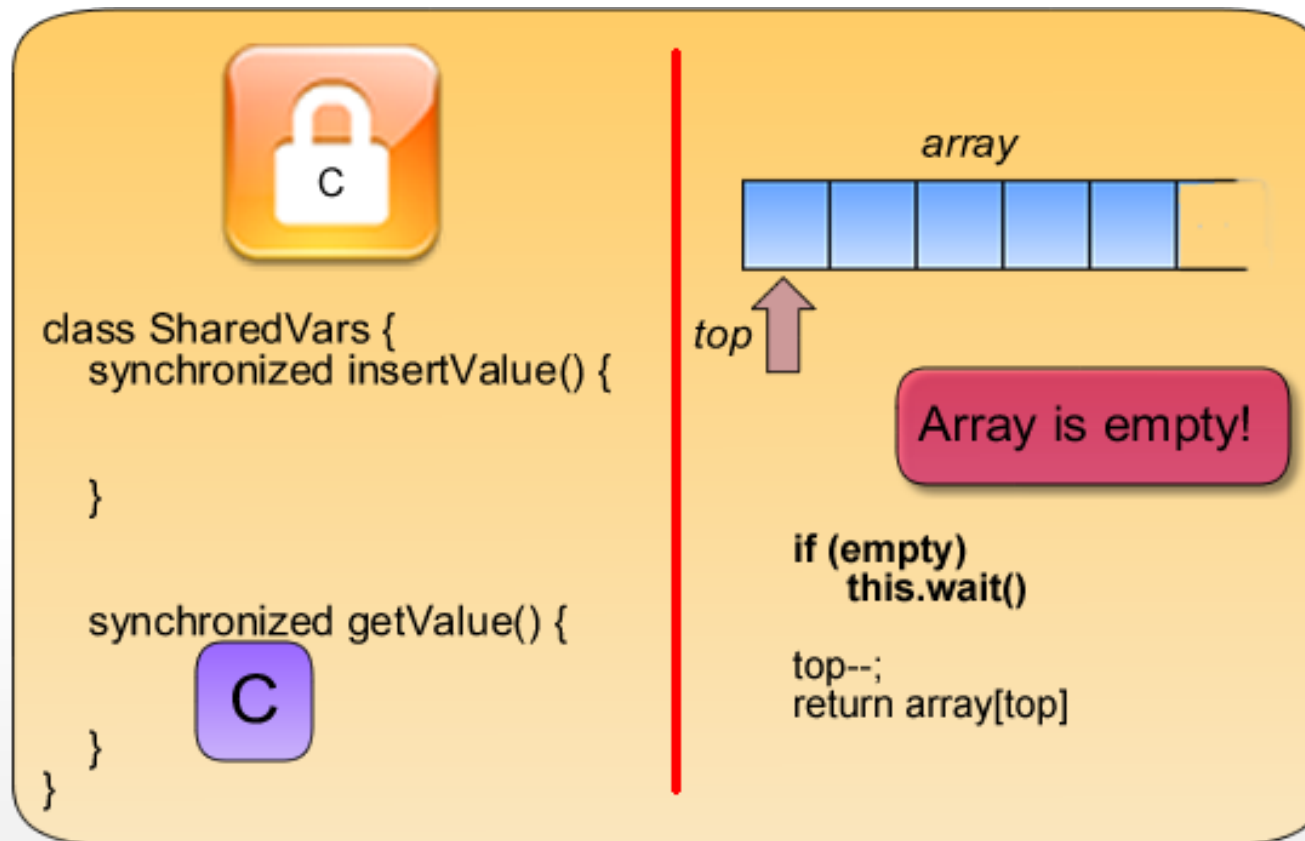
Wait() and notify()



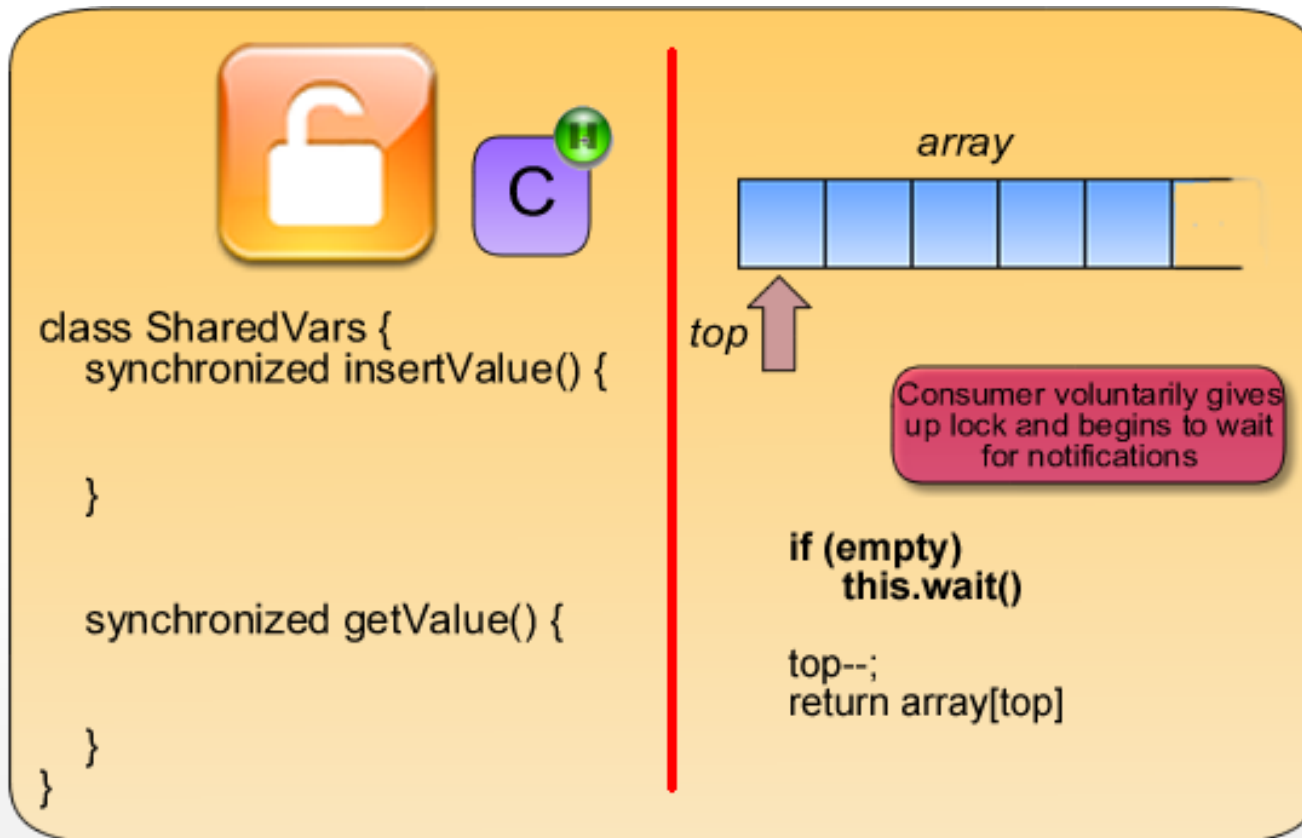
Wait() and notify()



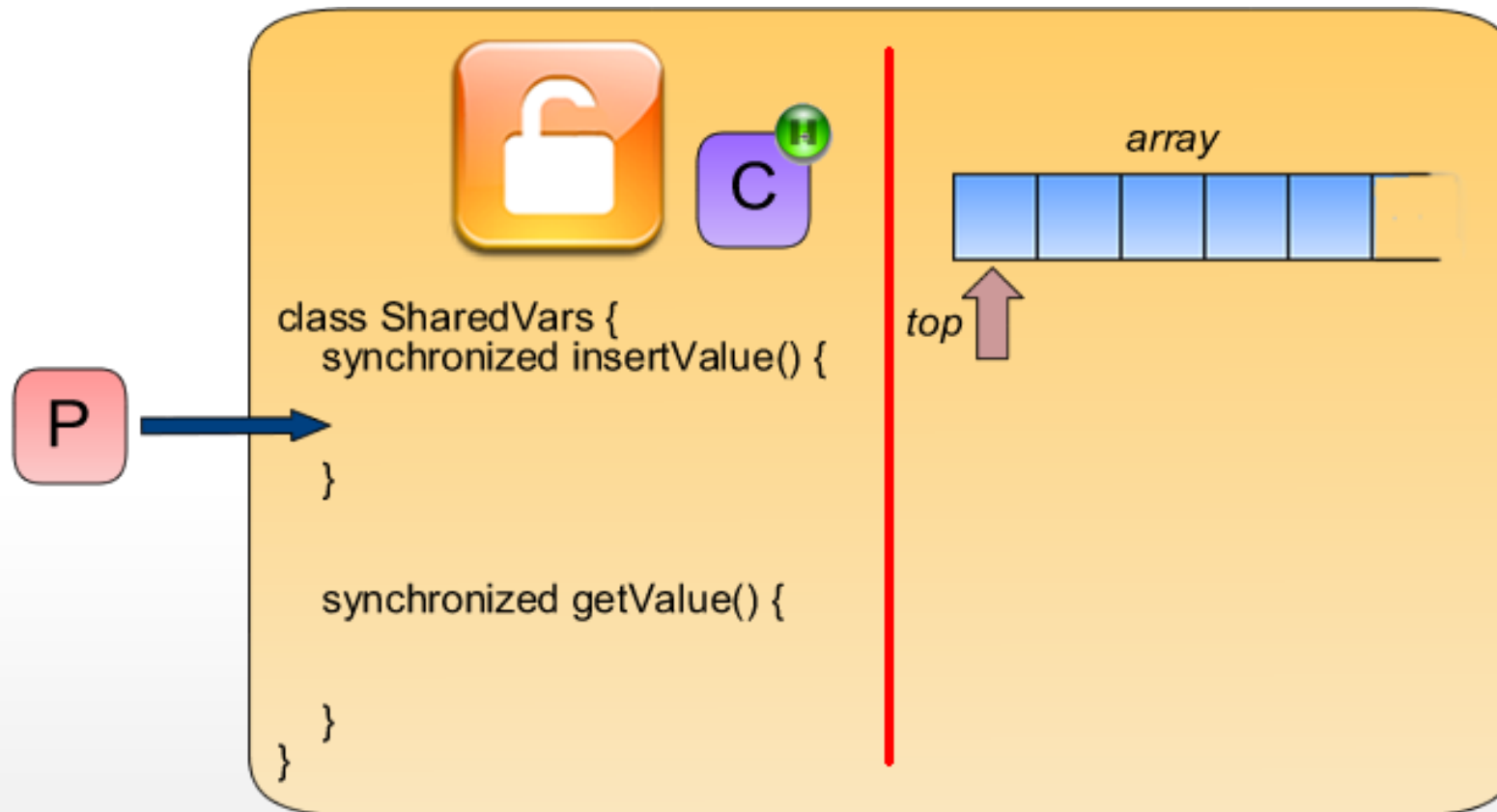
Wait() and notify()



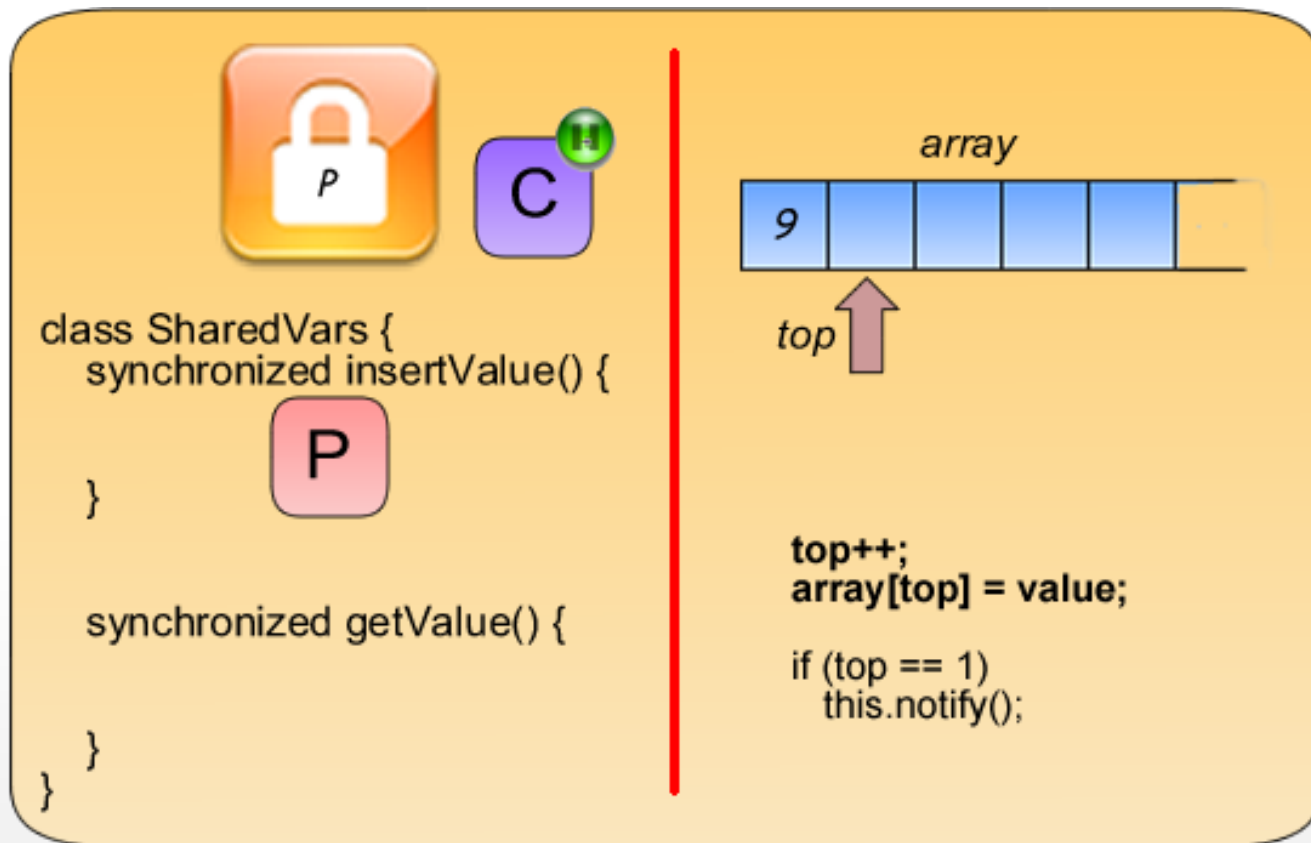
Wait() and notify()



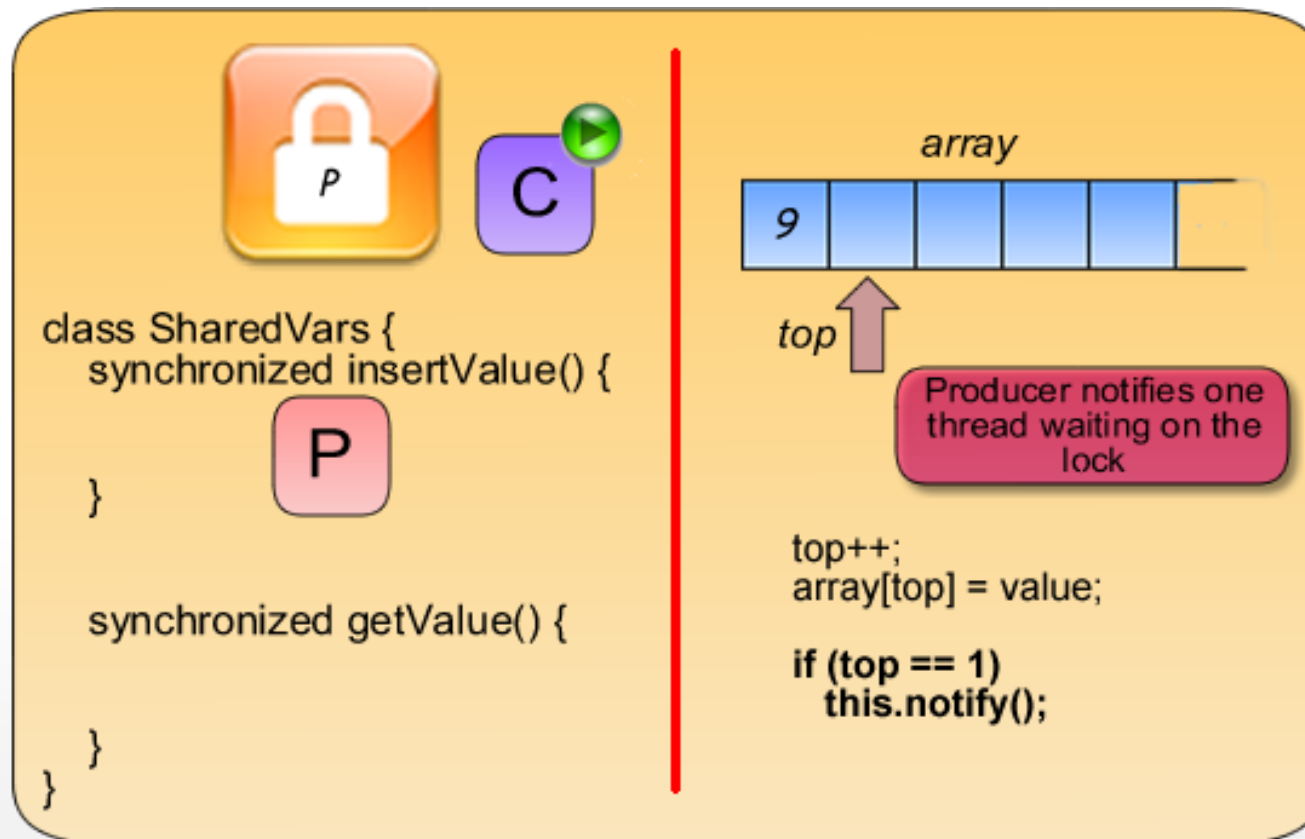
Wait() and notify()



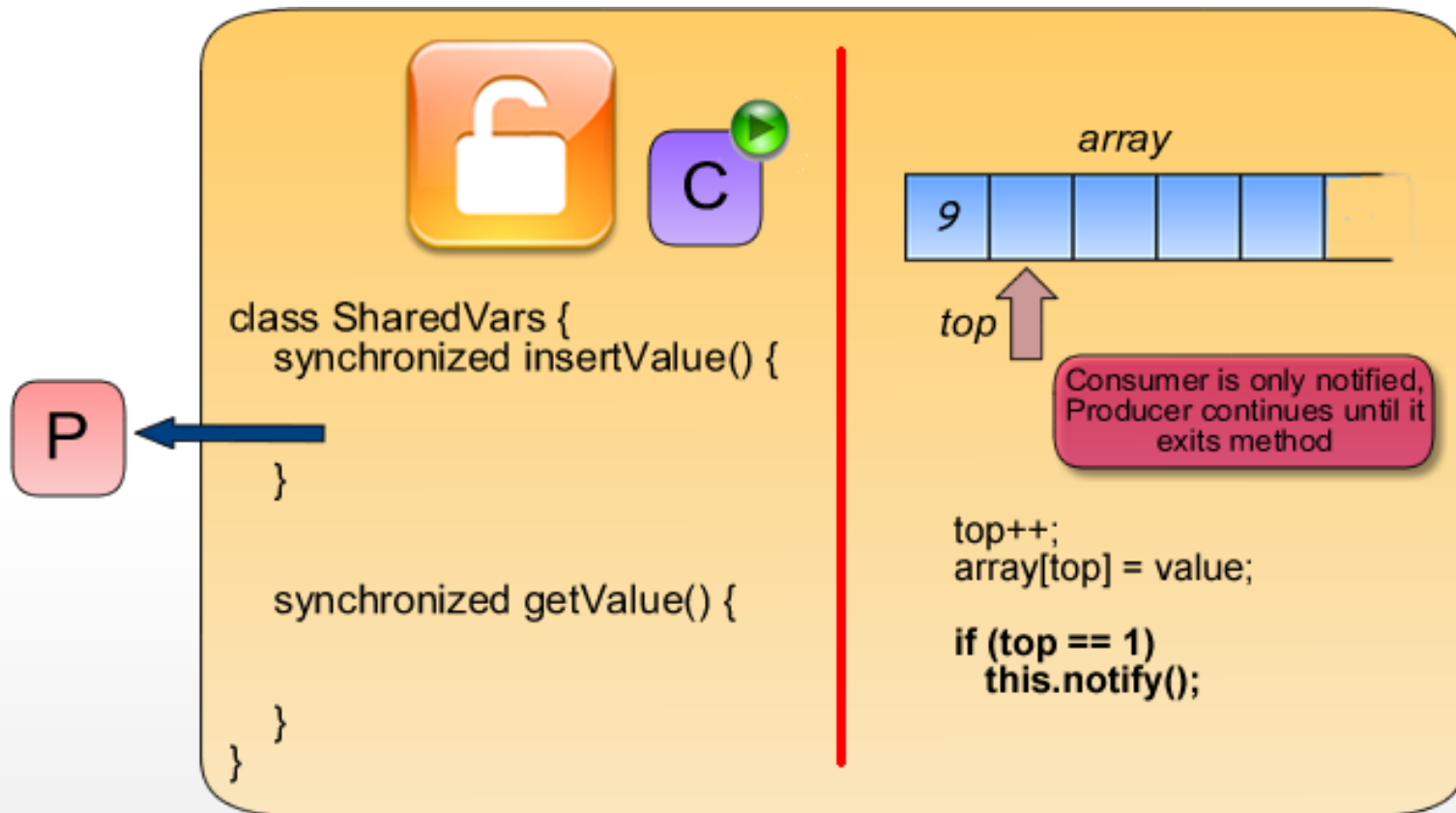
Wait() and notify()



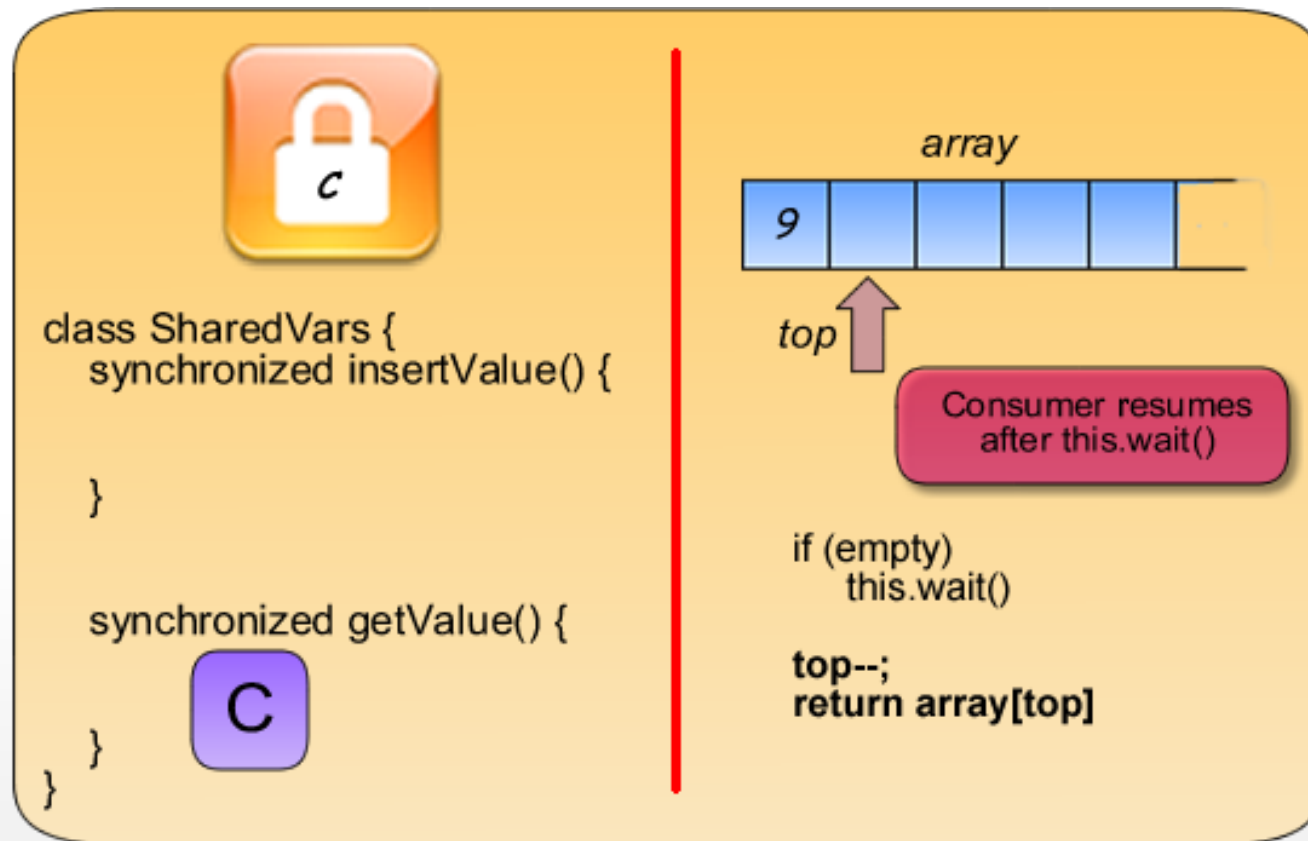
Wait() and notify()



Wait() and notify()



Wait() and notify()



Wait() and notify()

- Another method called notifyAll() notifies all the waiting threads.
- These waiting threads would then compete to see which single thread resumes execution, the rest of the threads would once again wait.

Producer-Consumer

- If you run the demo program again, you can see that, even if the producer is slow, the consumer waits until the producer gives out a value before getting that value
- Note that you must also consider inserting on a full array
 - Producer must wait until consumer has taken away some values before inserting
 - We will leave the implementation of this as an exercise

Wait() and notify()

- Note that this same thing must occur on inserting on a full array
 - Producer must wait until consumer has taken away some values before inserting
 - We will leave the implementation of this as an exercise.

Exercise

- Alice and Bob are at the office. There is a doorway that they have to pass through every now and then. Since Bob has good manners, if he gets to the door, he always makes sure that he always opens the door and waits for Alice to go through before he does. Alice by herself simply goes through the door.
- Your task is to write an Alice and Bob thread that mimics this behavior. To simulate having to pass through the door, have a delay in each thread for a random amount of time, maximum of 5 seconds. The output of your code should look something like this:

Alice: I go through the door

Alice: I go through the door

Bob: I get to the door and wait for Alice...

Alice: I go through the door

Bob: I follow Alice

Alice: I go through the door

Bob: I get to the door and wait for Alice