Java Threads

Outline

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

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

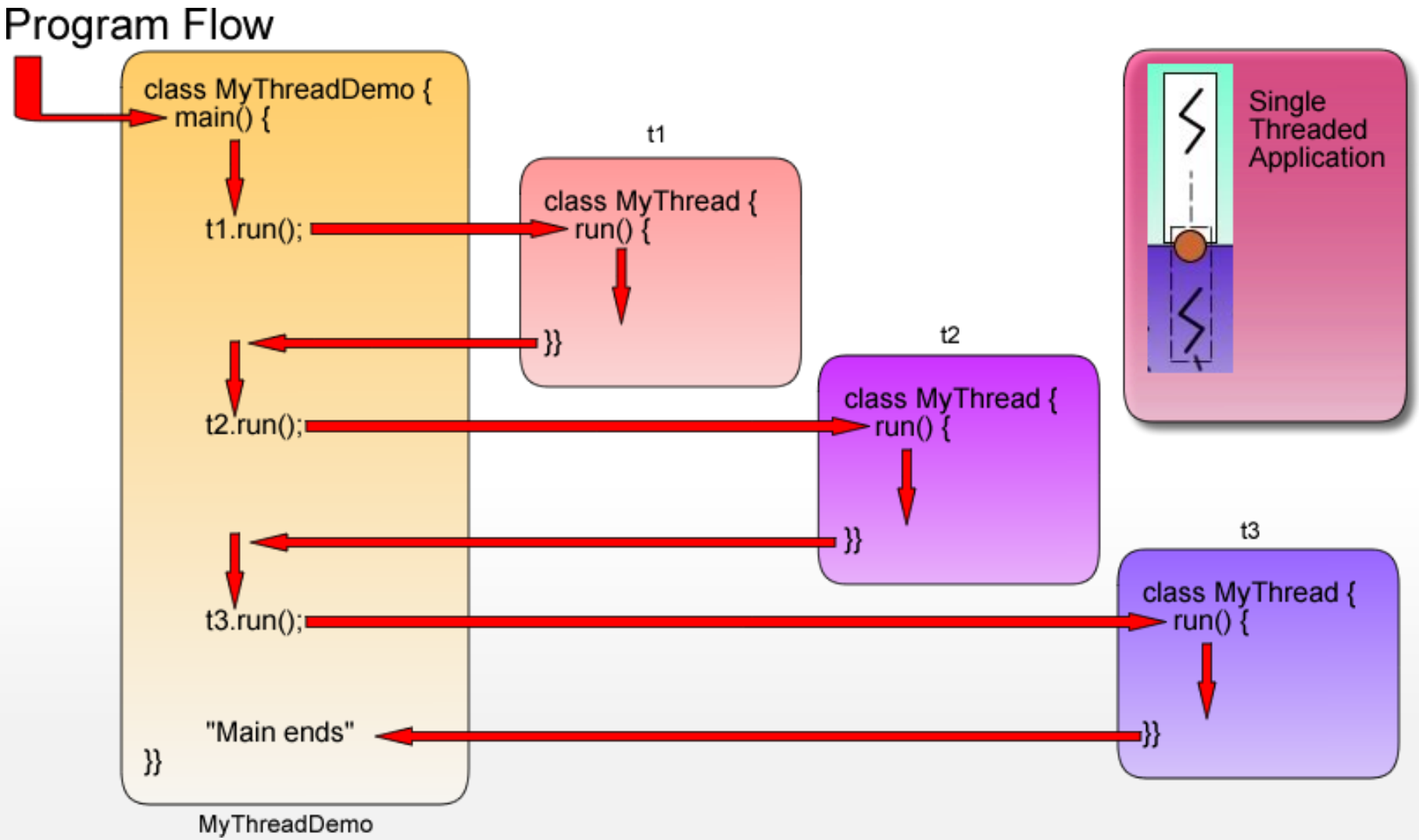
```
$ java MyThreadDemo
1111111111...22222222.....33333......Main ends
```

- 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 consecutuve 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



Basic threads

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

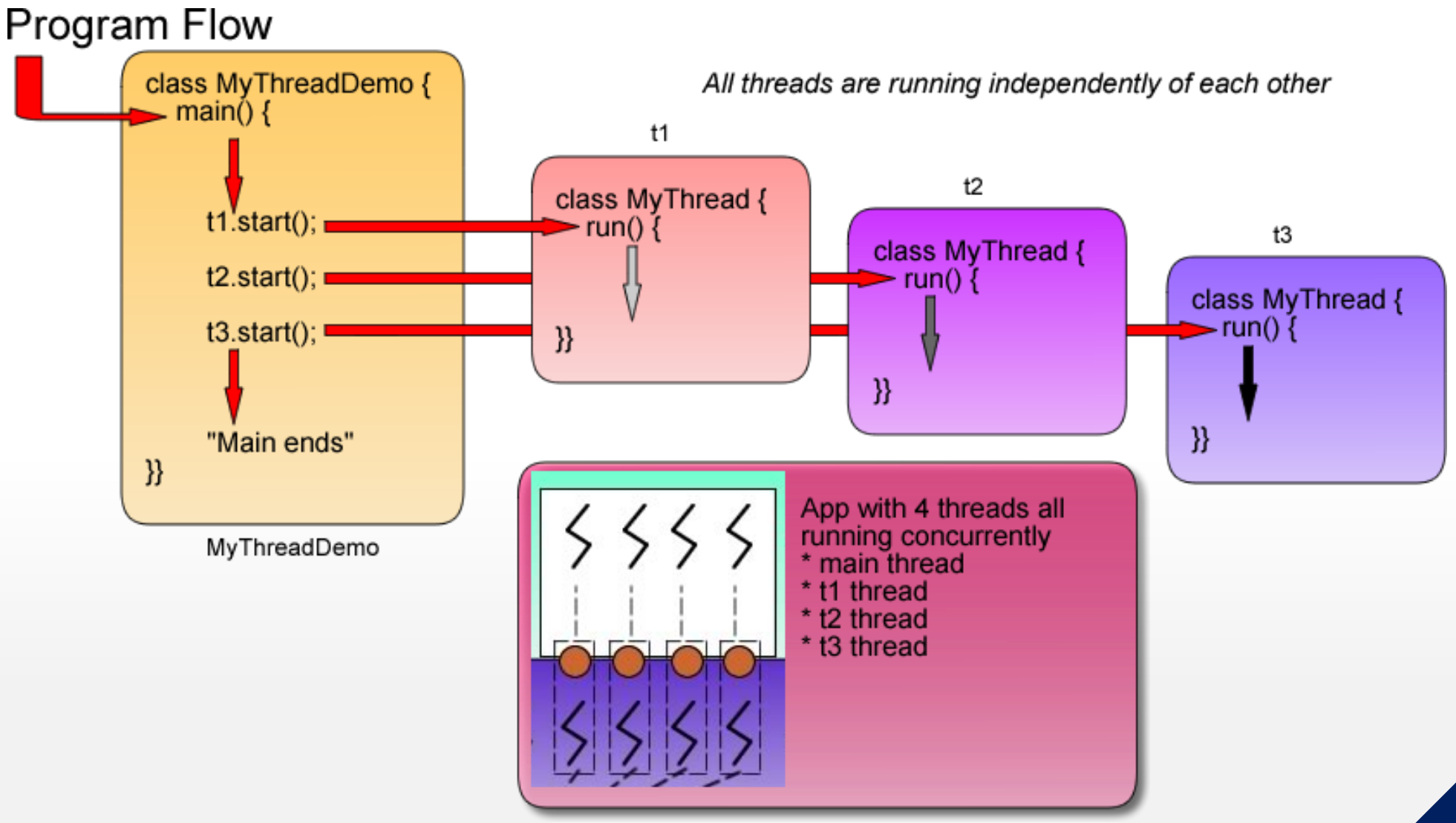
```
class MyThreadDemo {
        public static void main(String args[]) {
                MyThread t1 = new MyThread(1);
                MyThread t2 = new MyThread(2);
                MyThread t3 = new MyThread(3);
                t1.start();
                t2.start();
                t3.start();
                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
1111111111223311223211223333111Main ends111333222...
```

Basic threads



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.

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 miliseconds
        } 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 miliseconds
        } catch (InterruptedException e) { }
}
```

- The InterruptedException is an unchecked exeception 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.

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 occuring.
  - 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

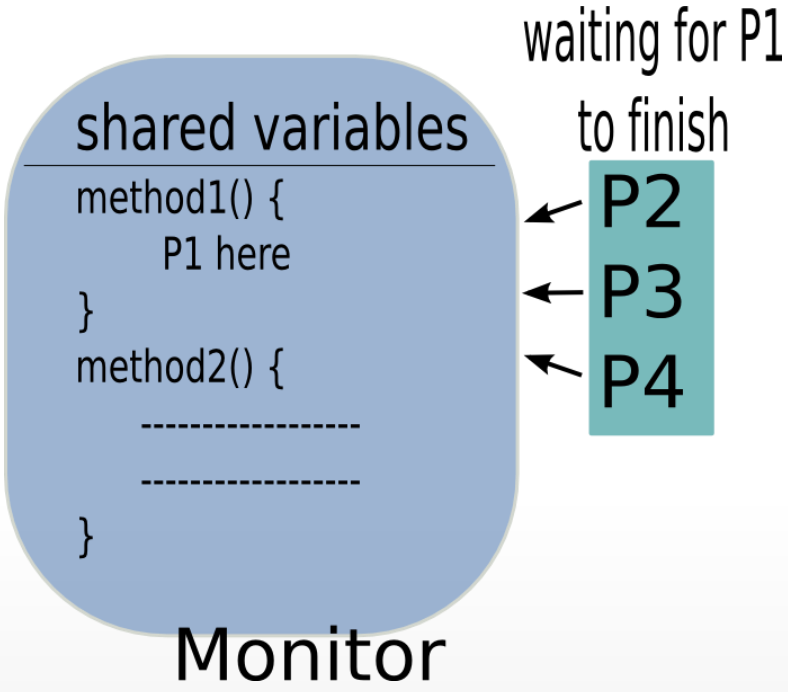
- To achieve our goal there should not be any context switches happening inside print10()
  - Only a single thread should be allowed to execute inside print10()
- As can be seen, this is an example of the critical section problem
- To solve this, we can use some of the techniques discussed in our synchronization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

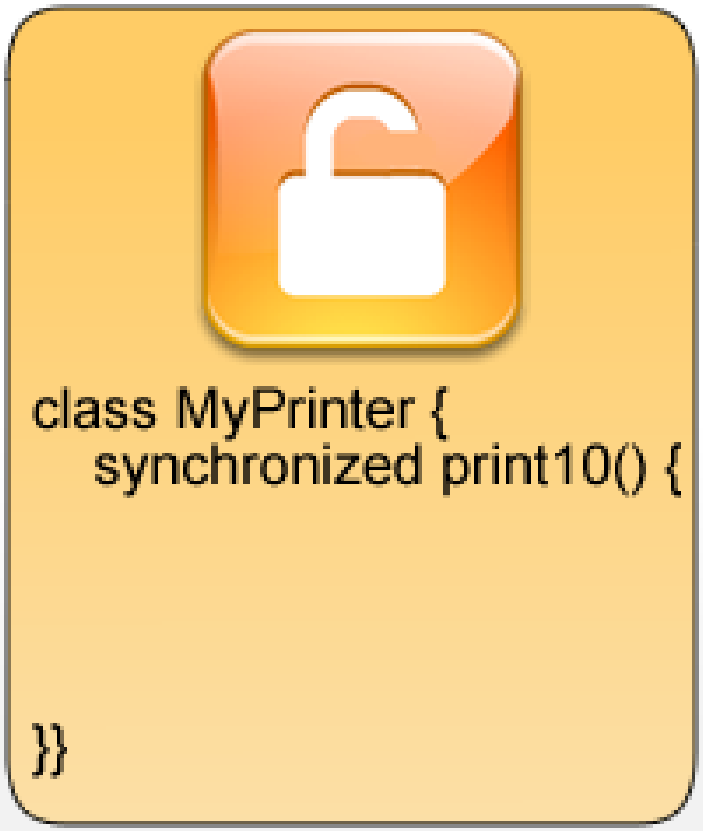
- However, even if we did turn our print10() method into a synchronized method, it will still not work

```
> 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

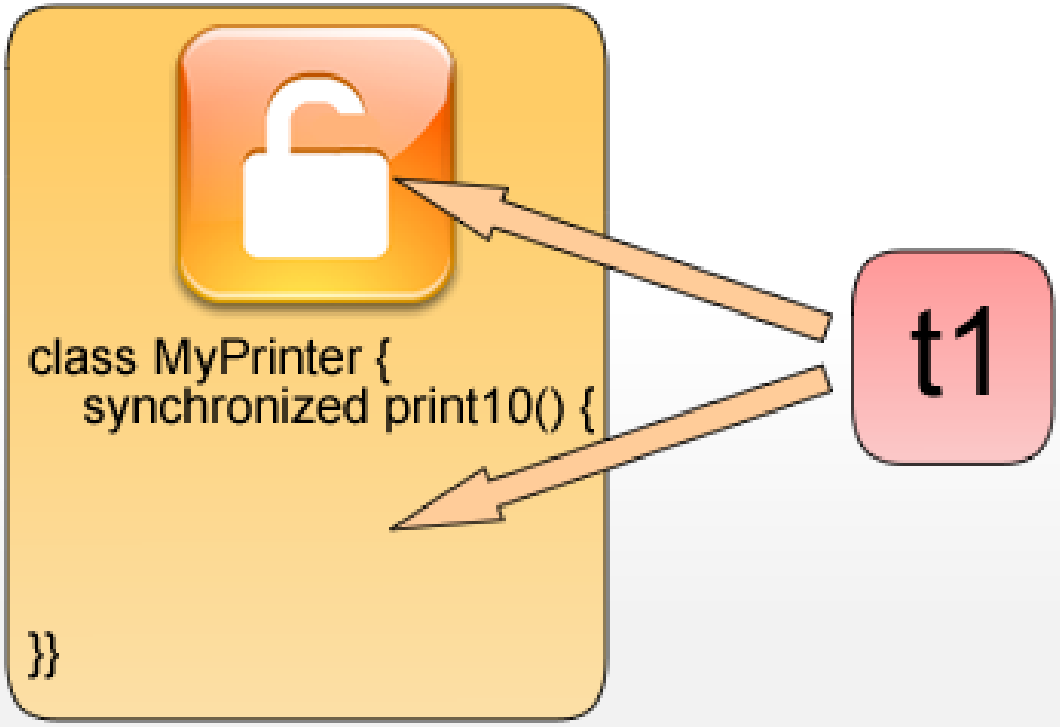
Intrinsic locks

- Every object in Java has an intrinsic lock



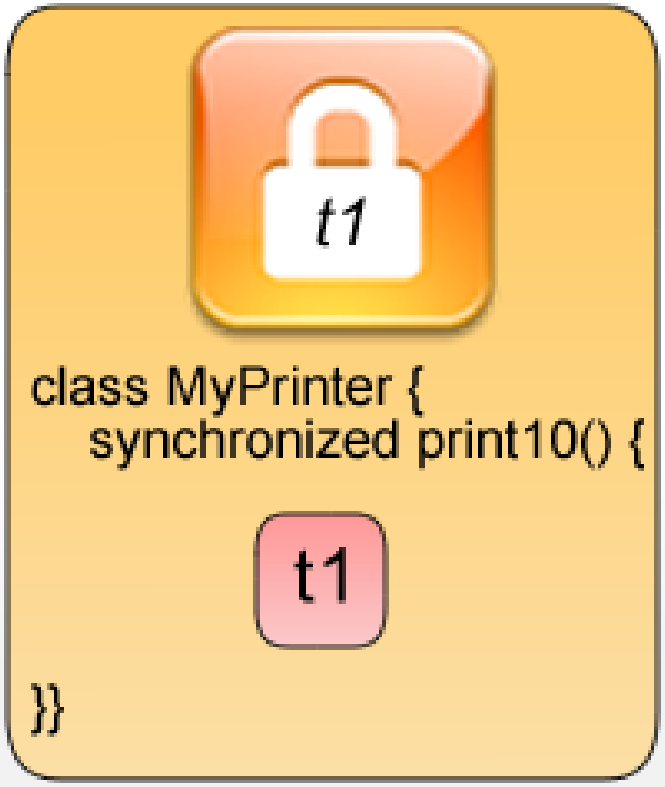
Intrinsic locks

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



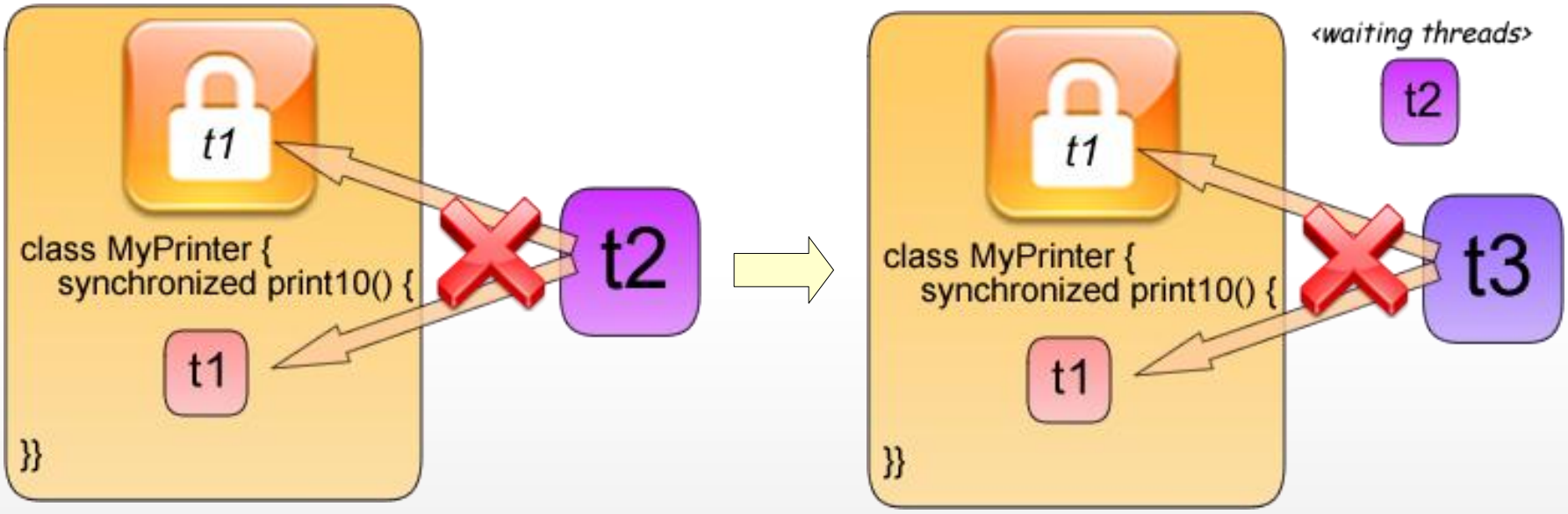
Intrinsic locks

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



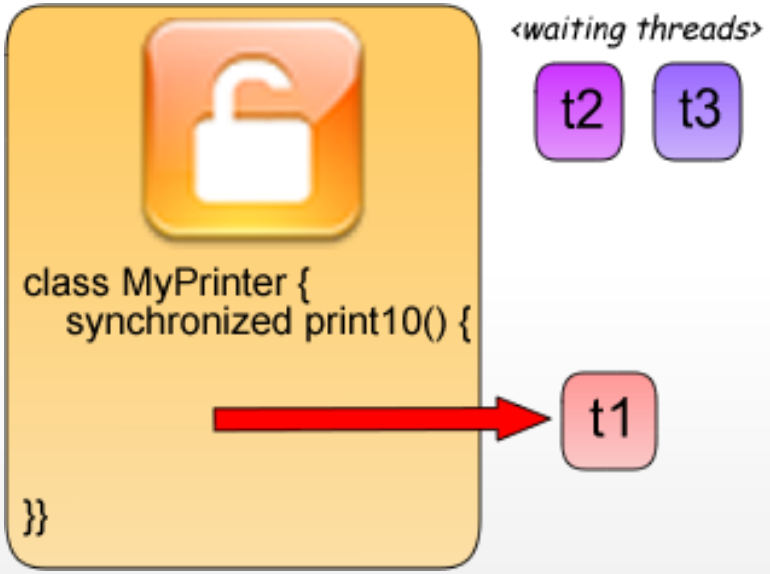
Intrinsic locks

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



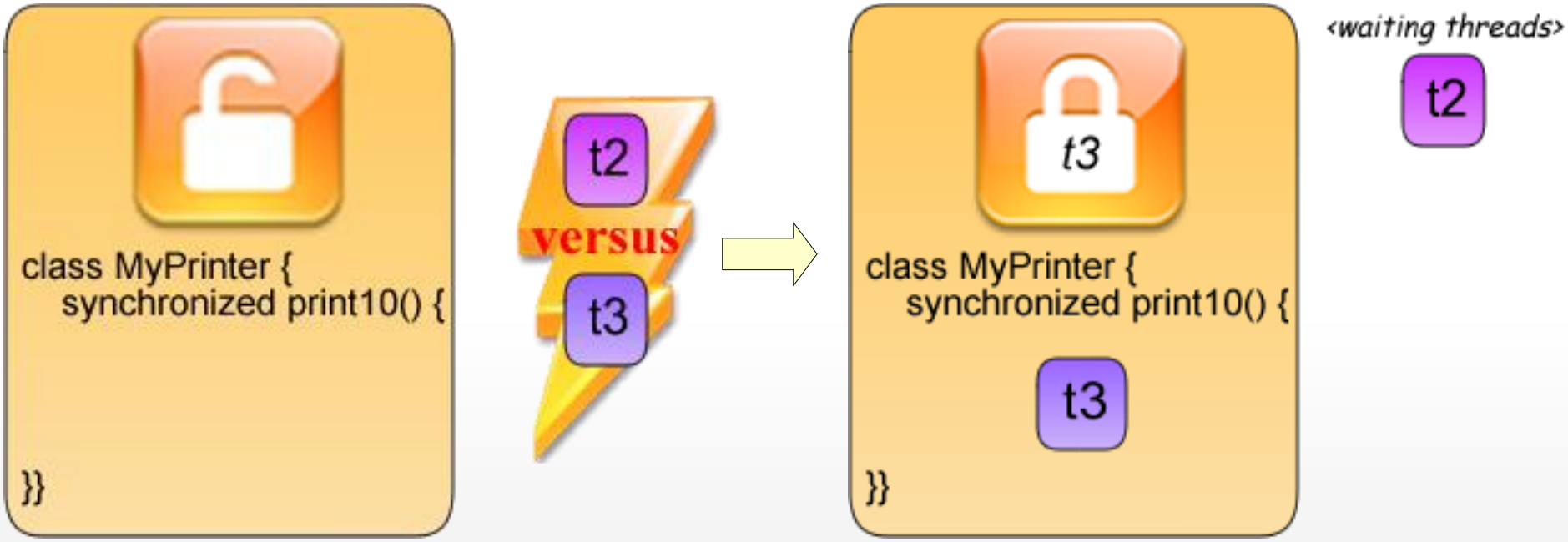
Intrinsic Locks

- 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

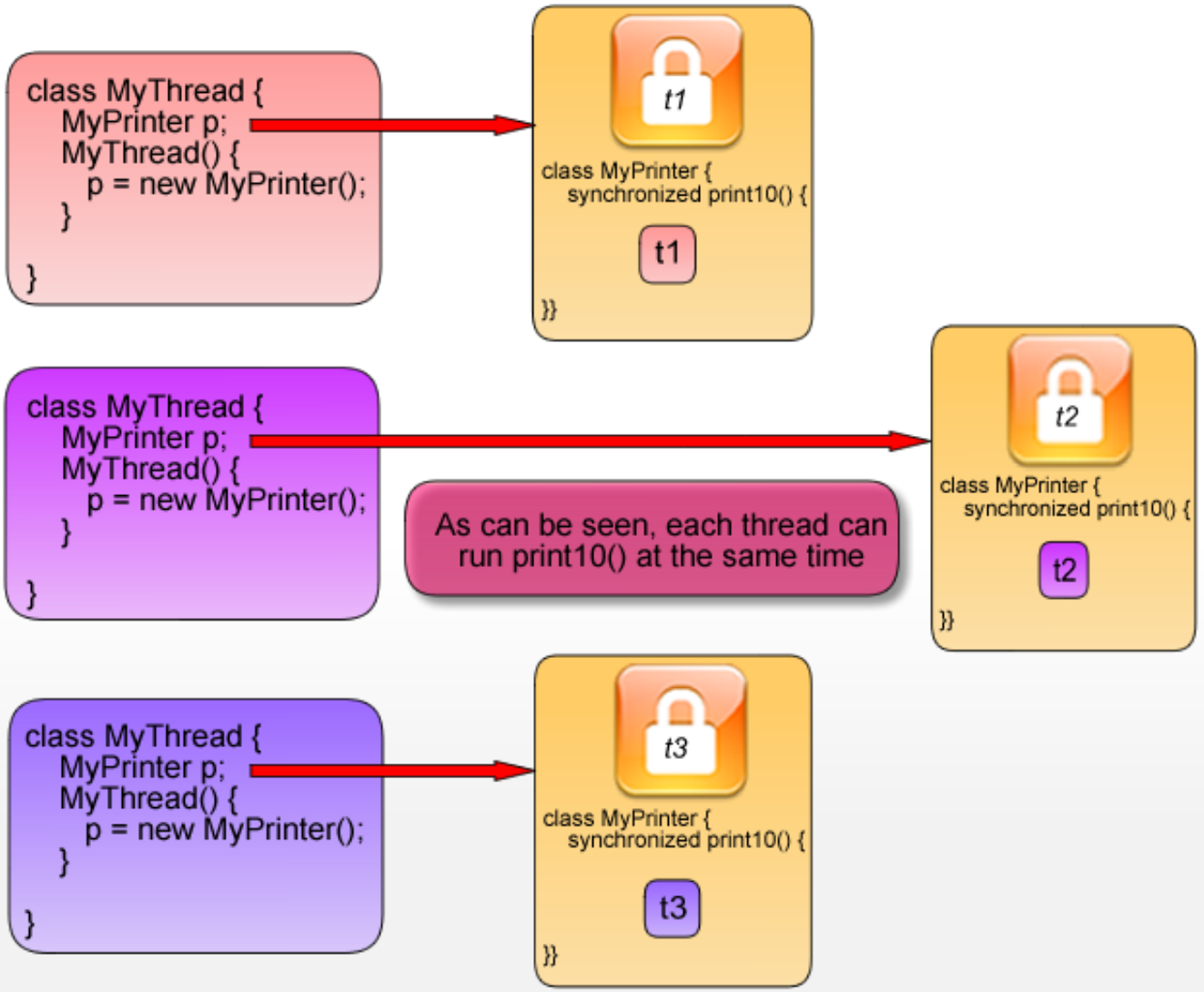


MyThreadDemo Failure

- 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

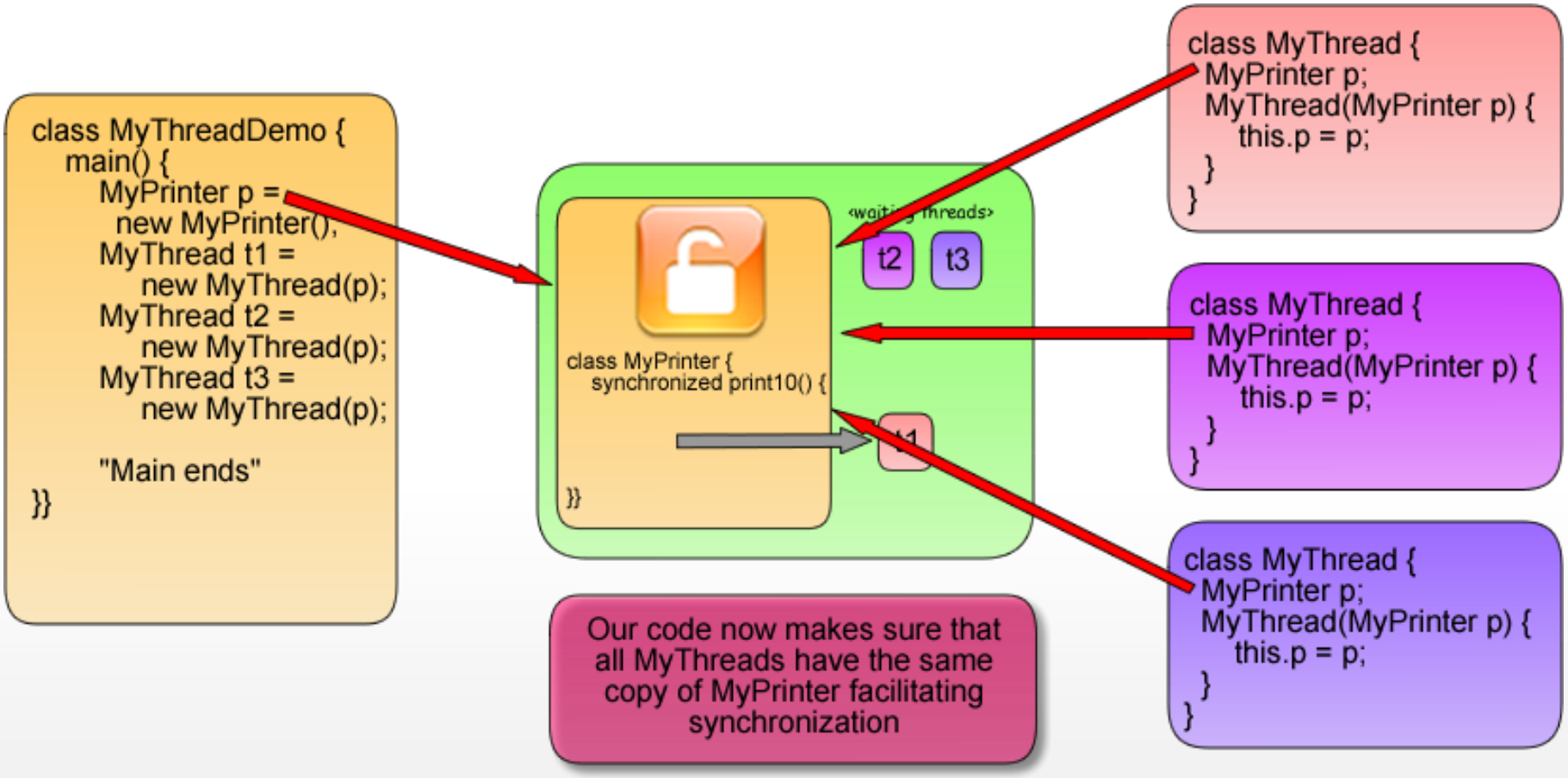
Solution

- 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


class MyThreadDemo {
main() {
MyPrinter p =
new MyPrinter();
MyThread t1 =
new MyThread(p);
MyThread t2 =
new MyThread(p);
MyThread t3 =
new MyThread(p);
"Main ends"
}}
class MyPrinter {
synchronized print10() {
}}
t2
t3
t1
class MyThread {
MyPrinter p;
MyThread(MyPrinter p) {
this.p = p;
}
}
class MyThread {
MyPrinter p;
MyThread(MyPrinter p) {
this.p = p;
}
}
class MyThread {
MyPrinter p;
MyThread(MyPrinter p) {
this.p = p;
}
}
Our code now makes sure that all MyThreads have the same copy of MyPrinter facilitating synchronization

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.

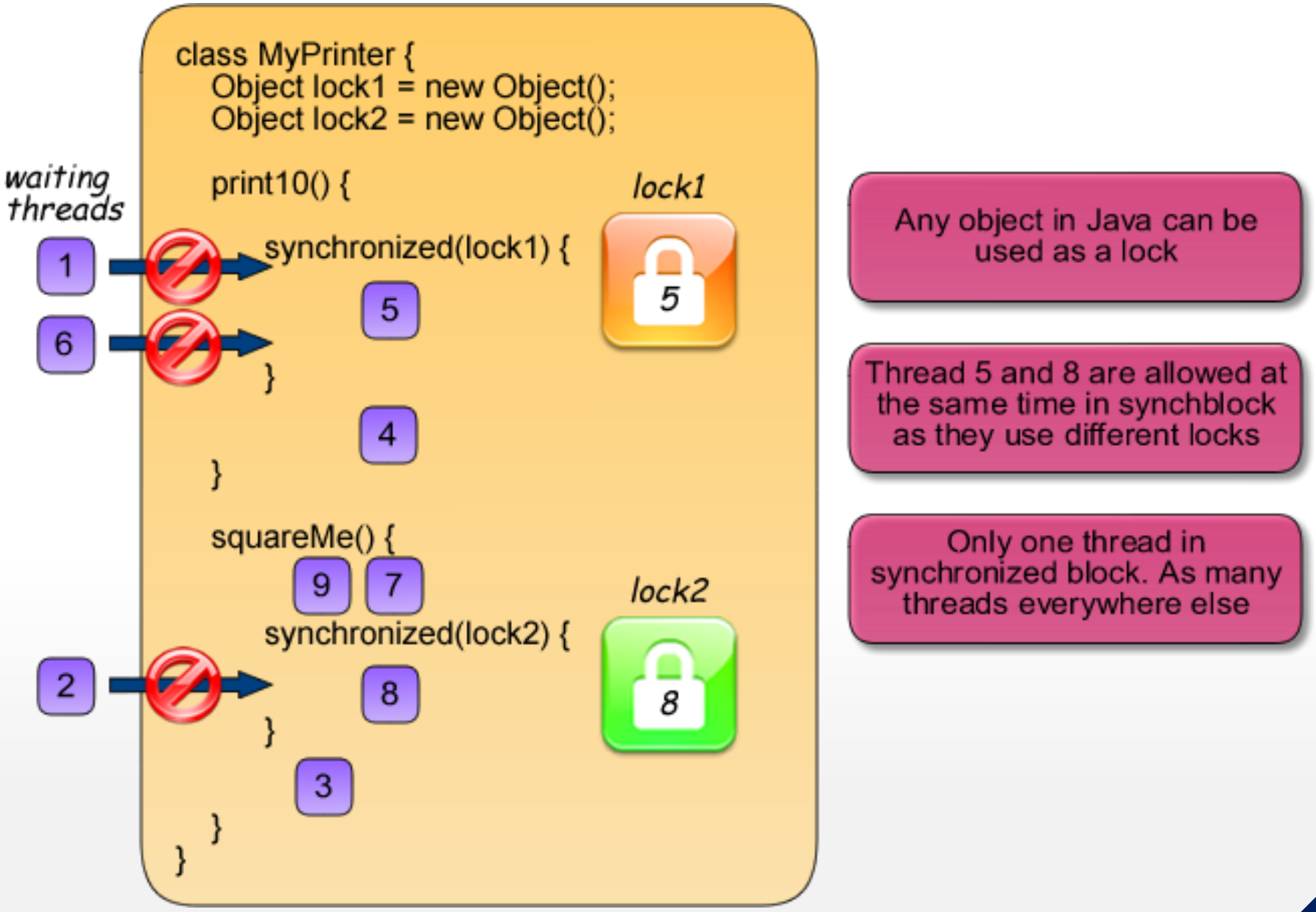
Multiple locks

- 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


class MyPrinter {
Object lock1 = new Object();
Object lock2 = new Object();
print10() {
synchronized(lock1) {
5
}
4
}
squareMe() {
9
7
synchronized(lock2) {
8
}
3
}
}
waiting threads
1
6
2
lock1
5
lock2
8
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

Synchronization/wait and notify

- **wait** is a method of the **Object** class. It causes the calling method to wait (blocked) until notified (when another thread calls **notify** or **notifyAll**)
- While waiting, a thread relinquishes its object locks. This gives other thread a chance to access the object.
- **notify** or **notifyAll** are all methods of the **Object** class.
  - **notify** randomly selects a thread among those waiting for an object lock (inside the object) and unblocks it.
  - **notifyAll** unblocks all threads waiting for an object lock (inside the object). Preferred because it reduces the probability of deadlock (Exercise: Why is this?).
- Note: **wait**, **notify** and **notifyAll** should only be called by a thread that is the owner of this object's monitor (within synchronized method).

Synchronization/**wait** and **notify**

- In our example, a thread calls **notifyAll** when it is done with a object.

```
public synchronized void transfer(int from, int to,
                                  int amount)
{  while (accounts[from] < amount)
   {  try {  wait(); }
      catch(InterruptedException e) {}
    }
   …
   notifyAll();
…}
```

SynchBankTest.java

Synchronization/Deadlock

Deadlock occurs when a number of threads waiting for each other

Example:

Account 1: $2,000
Account 2: $3,000
Thread 1: Transfer $3,000 from account 1 to account 2
Thread 2: Transfer $4,000 from account 2 to account 1

It is the responsibility of programmer to avoid deadlocks.

Summary

- A thread is a path of execution that executes within a process. When you run a Java program, the main() method runs in a thread. The main() method can start other threads.
- A process terminates when all its non-daemon threads run to completion.
- A thread is created by writing a class that implements java.lang. Runnable and associating an instance of this class with a new java.lang.Thread object. The new Thread is initially in the born state.
- Invoking the start() method on a Thread object starts the thread and places it in its appropriate runnable queue, based on its priority.

Summary

- Java uses a preemptive scheduling mechanism where threads of higher priority preempt running threads of a lower priority. However, the actual behavior of threads also relies on the underlying platform.

- A thread can also be written by writing a class that extends java.util.TimerTask and associating an instance of this class with a java.util.Timer object. This type of thread is useful when performing scheduled tasks.

Summary

- The synchronized keyword is used to synchronize a thread on a particular object. When a thread is synchronized on an object, the thread owns the lock of the object. Any other thread attempting to synchronize
- Care needs to be taken when synchronizing threads, since deadlock may occur. Ordering locks is a common technique for avoiding deadlock.
- The wait() and notify() methods from the Object class are useful when multiple threads need to access the same data in any type of producer/consumer scenario.