Threads

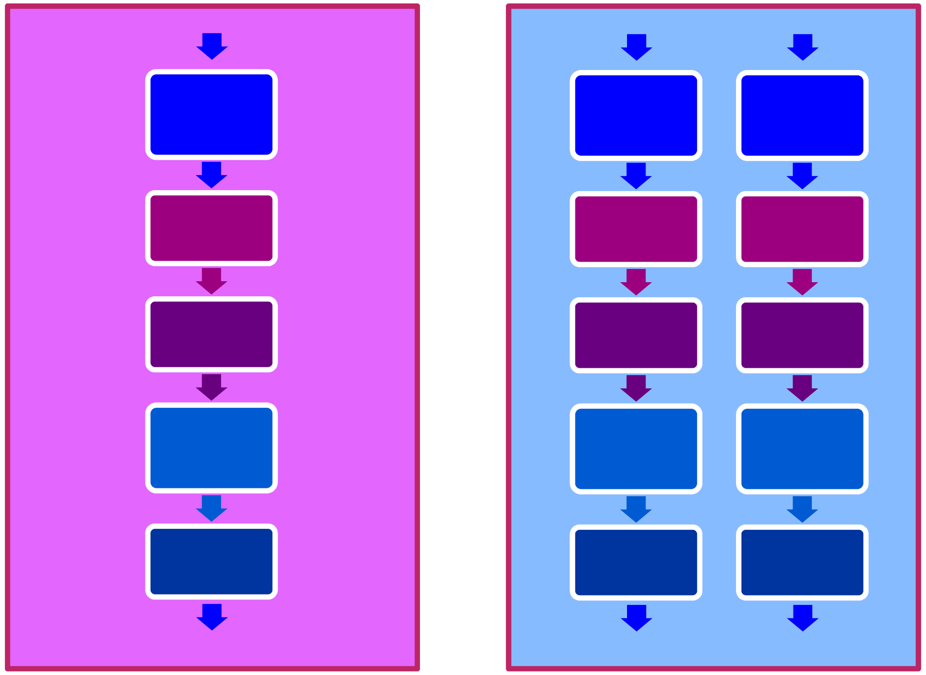
Chate Patanothai

Objectives

- Knowing thread: 3W1H
- Create separate threads
- Control the execution of a thread
- Communicate between threads
- Protect shared data

What are threads?

- An execution context
  - a virtual CPU
  - the code for executing
  - the data



- A process is a program in execution
- A process has one or more threads

Thread code and data

- In Java, the virtual CPU is encapsulated in an instance of the `Thread` class
- Two threads share the **same code** when they execute from instances of the same class
- Two threads share the **same data** when they share access to a common object

Making a thread

- New class `extends Thread`
  - simple
  - cannot extend from other class
- Creating a new class that `implements Runnable` interface (preferred)
  - better OOD
  - single inheritance
  - consistency
- Overriding `run()` method

Creating the thread

- create an instance of **`Runnable`**
- the **`Thread`** class already implemented **`Runnable`** interface

Starting the Thread

- Using the `start` method
- Placing the thread in *runnable* state

Subclass of Thread

```java
public class SomeThread extends Thread {

  public void run() {
    // code for thread execution

  }

}
```

```java
public class ThreadTester {
  public static void main(String[] args) {
    // creating a thread
    SomeThread t = new SomeThread();

    // start the thread
    t.start();
  }
}
```

Implementing Runnable

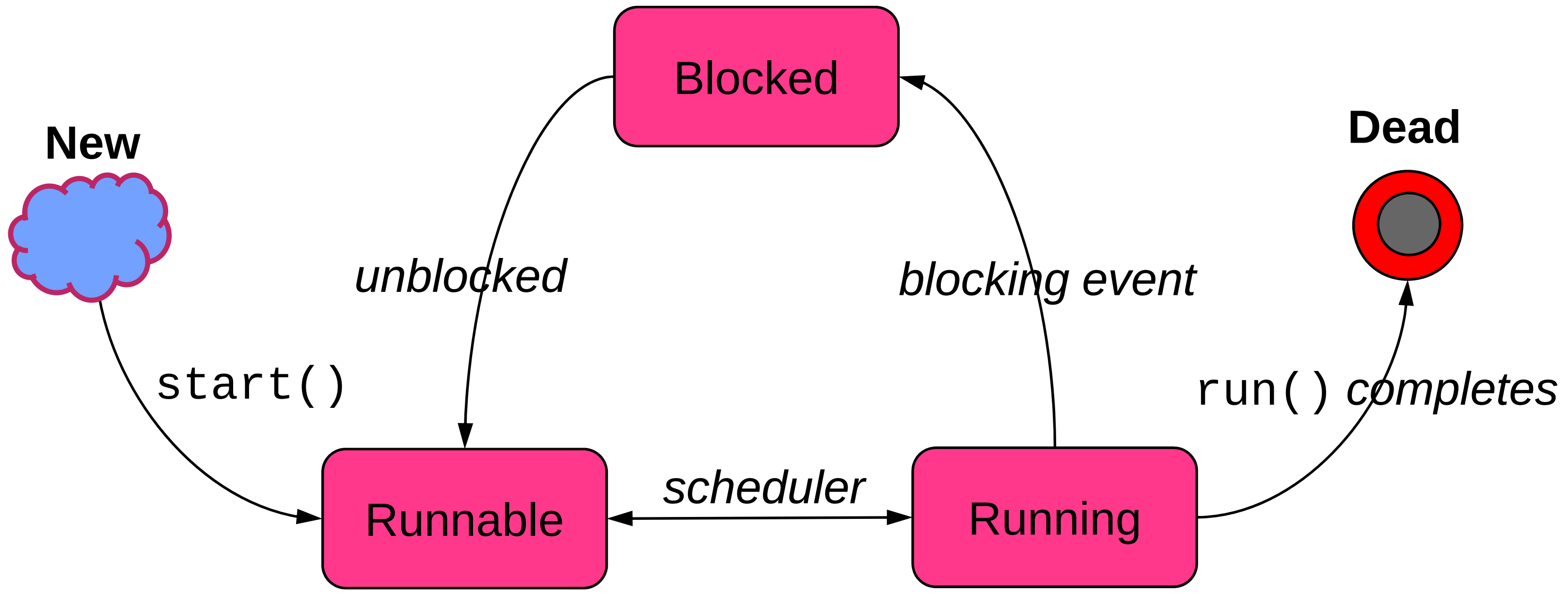
```
public class RunningClass [extends XXX] implements Runnable {

  public void run() {  // must be overridden
    // code for thread execution
  }
}
```

```
public class ThreadTester {
  public static void main(String[] args) {
    // creating an instance of a Runnable
    RunningClass rc = new RunningClass();

    // creating a new thread for the Runnable instance
    Thread t = new Thread(rc);

    // starting the thread
    t.start();
  }
}
```

Basic Thread States

```
Thread t = new Thread();
```

```
t.start();
```



Sleeping (ZZZzzzzzz)

- allow other threads a chance to execute
- *sleep* is a *static* method in the **`Thread`** class
- throws **`InterruptedException`**

```
public class Runner implements Runnable {
  public void run() {
    while (true) {
      // do lots of interesting stuff
       :
      // Give other threads a chance
      try {
        Thread.sleep(10);  // time in milliseconds
      } catch (InterruptedException e) {
        // This thread’s sleep was interrupted by another thread
      }
    }
  }
}
```

Terminating a Thread

- when a thread completes, it *cannot* run again
- using a flag to indicate the exit condition

```
public class Runner implements Runnable {
  private boolean done = false;
  public void run() {
    while (!done) {
        . . .
    }
  }

  public void stopRunning() {
    done = true;
  }
}
```

```
public class ThreadController {
  private Runner r = new Runner();
  private Thread t = new Thread(r);

  public void startThread() {
    t.start();
  }

  public void stopThread() {
    r.stopRunning()
  }
}
```

Basic Control of Threads

- Testing threads:
  - `isAlive()`
- Accessing thread priority:
  - `getPriority()`
  - `setPriority()`
- Putting threads on hold:
  - `Thread.sleep()`
  - `join()`
  - `Thread.yield()`

Thread Priority

- `Thread.MIN_PRIORITY` (1)
- `Thread.NORM_PRIORITY` (5)
- `Thread.MAX_PRIORITY` (10)

The `join` Method

- wait until the thread on which the **`join`** method is called terminates

```
public static void main(String[] args) {
  Thread t = new Thread(new Runner());
  t.start();
  . . .
  // do stuff in parallel
  . . .
  // wait for t to finish
  try {
    t.join();
  } catch (InterruptedException e) {
    // t came back early
  }
  // continue this thread
  . . .
}
```

The `Thread.yield` Method

- give other *runnable* threads a chance to execute
- places the calling thread into the *runnable* pool if there are thread(s) in *runnable*,
- if not, **`yield`** does nothing
- **`sleep`** gives lower priority threads a chance
- **`yield`** gives other *runnable* threads a chance

Shared data

```
public class MyStack {
  int idx = 0;
  char[] data = new char[6];

  public void push(char c) {
    data[idx] = c;
    idx++;
  }

  public char pop() {
    idx--;
    return data[idx];
  }
}
```

- one thread (A) pushing data onto the stack
- one thread (B) popping data off the stack

buffer	p	q				
idx = 2			^			

A just finished push a character, then preempted

buffer	p	q	**r**			
idx = 2			**^**			

B is now in Running

The Object Lock Flag

- Every object has a “lock flag”
- use `synchronized` to enable interaction with this flag

Using `synchronized`

```
public class MyStack {
  . . .
  public void push(char c) {
    synchronized(this) {
      data[idx] = c;
      idx++;
    }
  }
  . . .
}
```

```
  public void push(char c) {
    synchronized(this) {
      data[idx] = c;
      idx++;
    }
  }
```

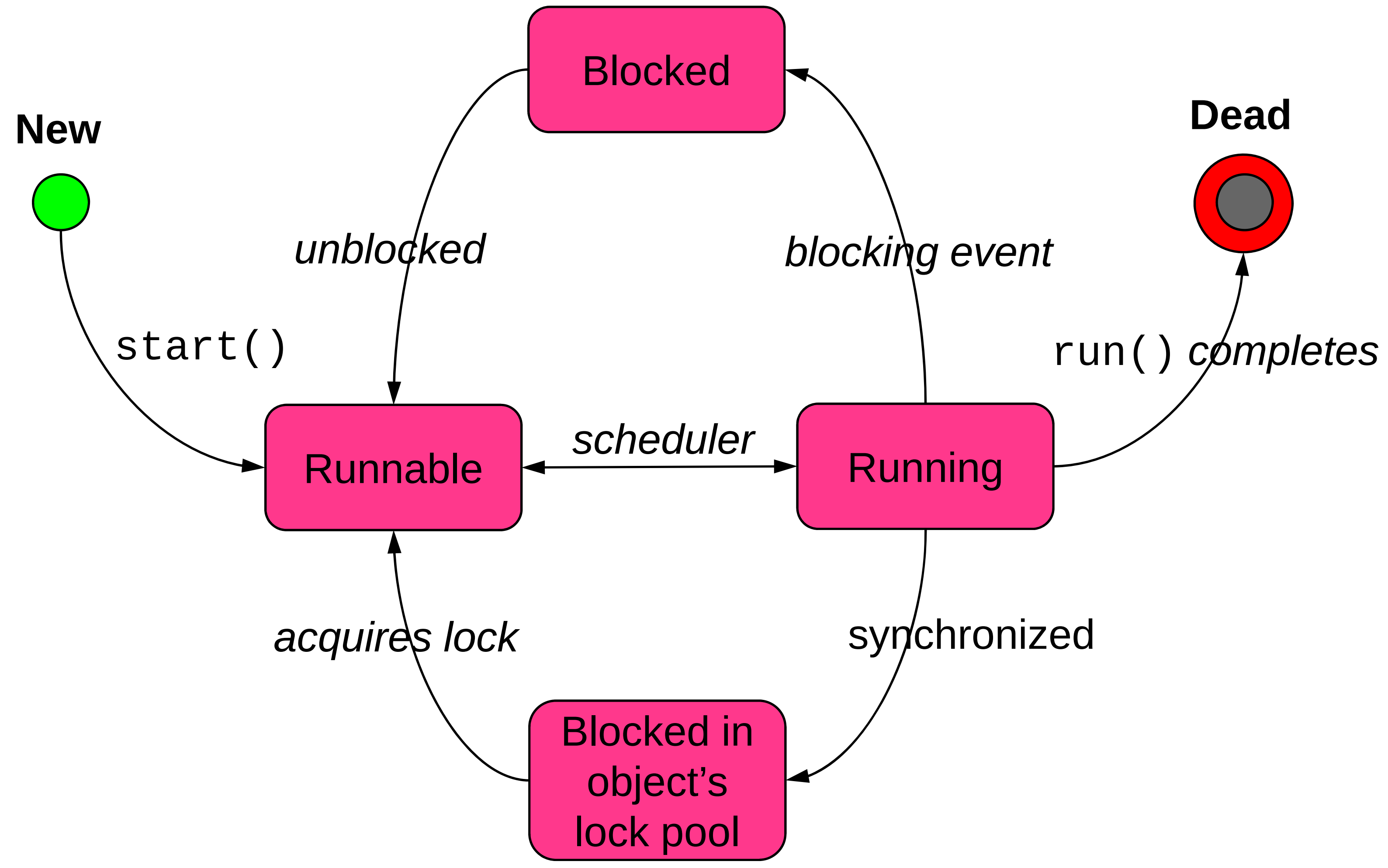
Releasing the Lock Flag

- A thread waiting for the lock flag of an object cannot resume running until it get the flag
- Released when the thread passes the end of the **`synchronized`** code block
- Automatically released when a break, return, or exception is thrown by the **`synchronized`** code block

Shared Data

- All access to shared data should be **`synchronized`**
- Shared data protected by **`synchronized`** should be **`private`**

Thread States (synchronized)



Deadlock

- Two threads waiting for a lock from other

Thread A locks 🕮, and waits for 🕮

Thread A locks 🕮, and waits for 🕮

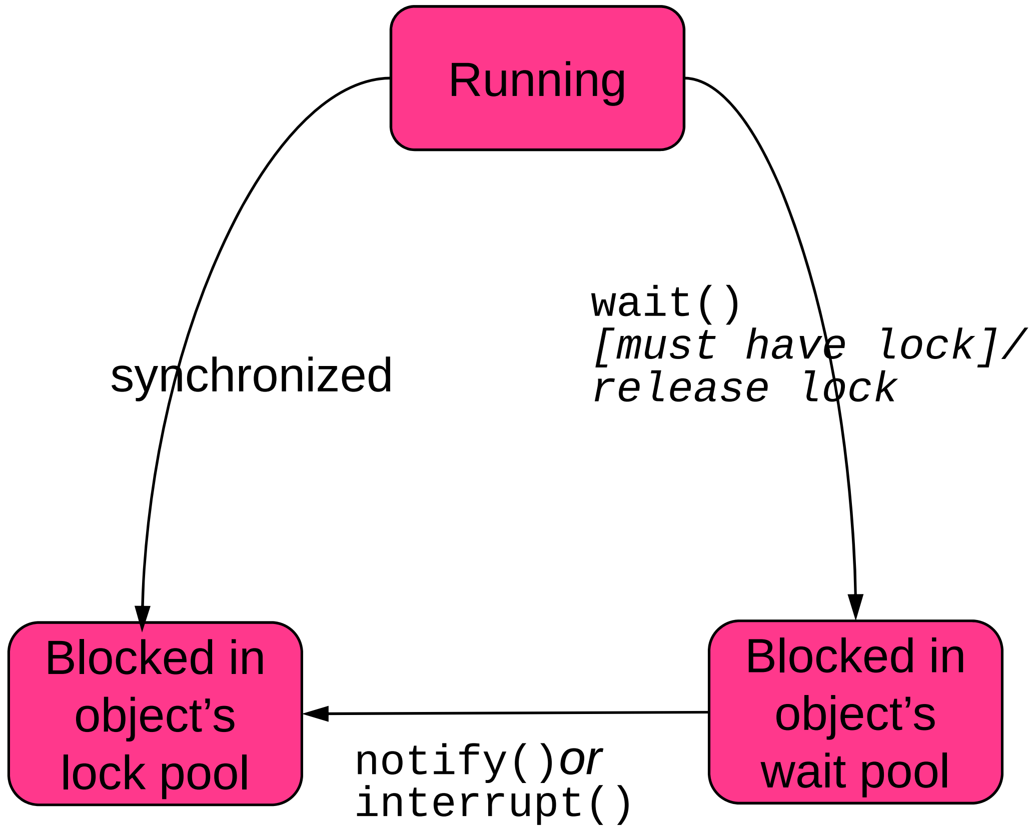
- no detection or avoidance by Java
- Can be avoided by
  - the order to obtain locks
  - applying the order throughout the program
  - releasing the lock in the reverse order

Thread Interaction

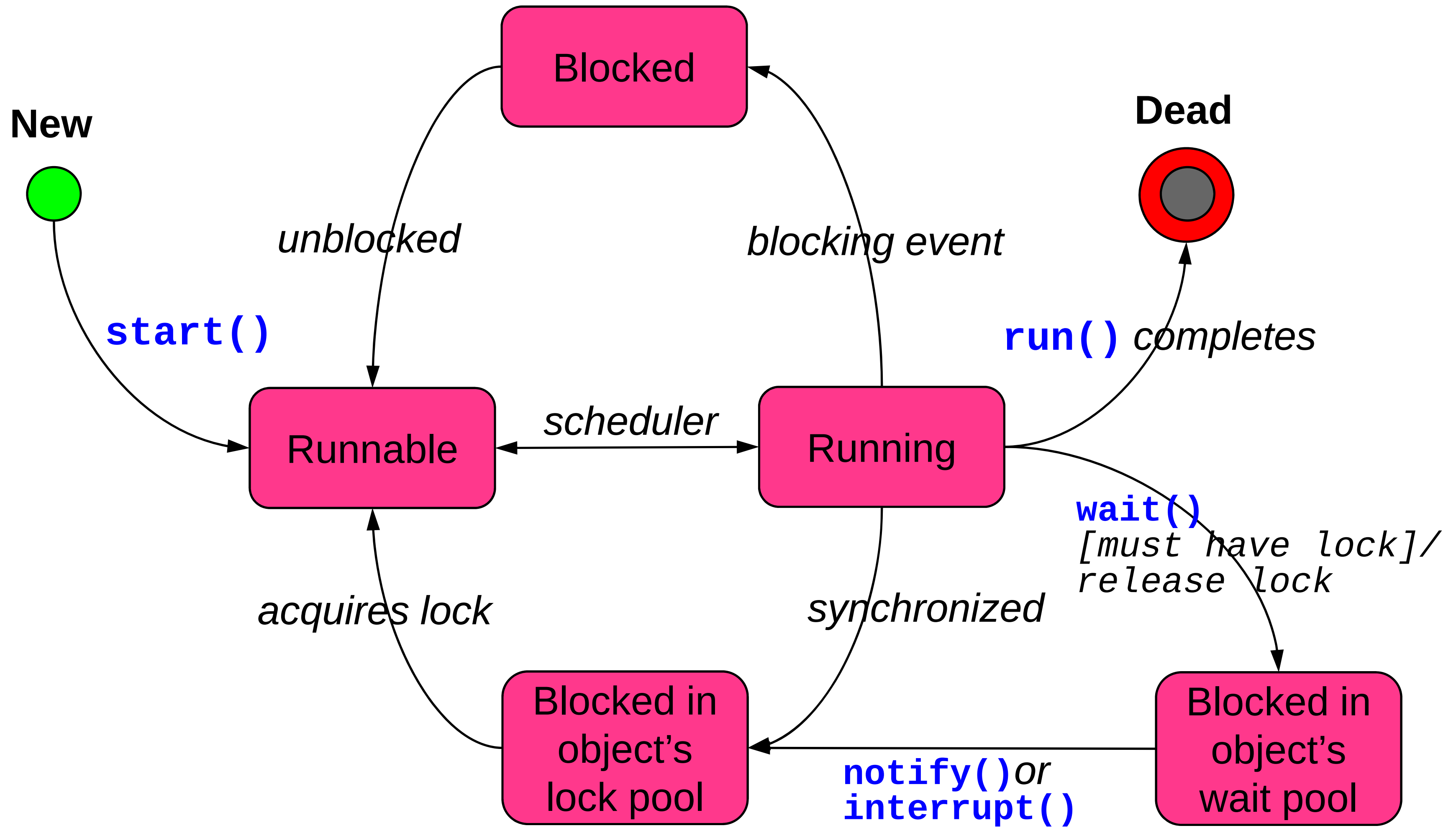
- `wait` and `notify`
- methods from `java.lang.Object`
- if a thread issues a `wait` call on an object x, it pauses its execution until another thread issues a `notify` call on the same object x
- the thread MUST have the lock for that object (`wait` and `notify` are called only from within a synchronized block on the instance being called)

The pools

- Wait pool
  - execute **`wait()`**
- Lock pool
  - thread moved from wait pool
  - **`notify()`**
    - arbitrary thread
  - **`notifyAll()`**
    - all threads



Thread States (wait/notify)


New
Dead
Blocked
unblocked
blocking event
start()
run() completes
Runnable
scheduler
Running
wait()
[must have lock]/
release lock
acquires lock
synchronized
Blocked in object's lock pool
notify()or
interrupt()
Blocked in object's wait pool