Chapter 32 Multithreading

Objectives

To explain multithreading

To develop task classes by implementing the Runnable interface

To create threads to run tasks using the Thread class

To control threads using the methods in the Thread class

To control animations using threads

To run code in the event dispatch thread

To execute tasks in a thread pool

To use synchronized methods or blocks to synchronize threads to avoid race conditions

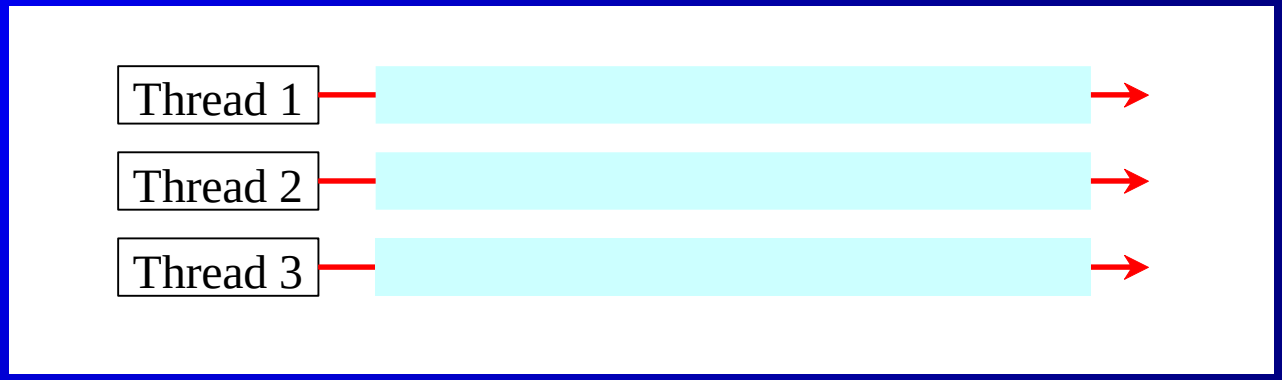
To synchronize threads using locks

To facilitate thread communications using conditions on locks

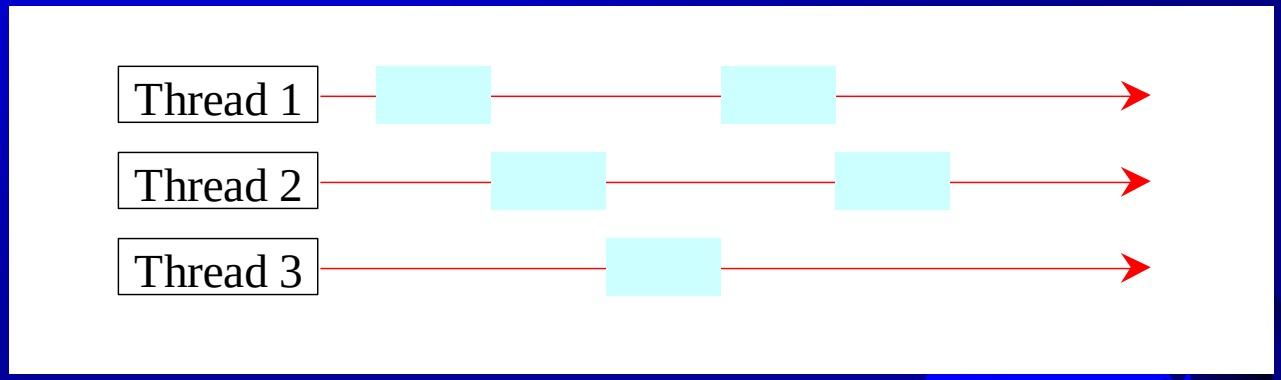
To describe the life cycle of a thread

Threads Concept

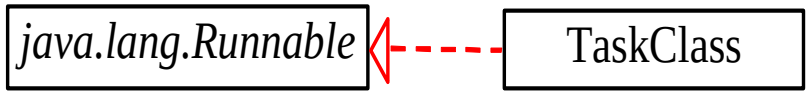
Multiple threads on multiple CPUs



Multiple threads sharing a single CPU



Creating Tasks and Threads



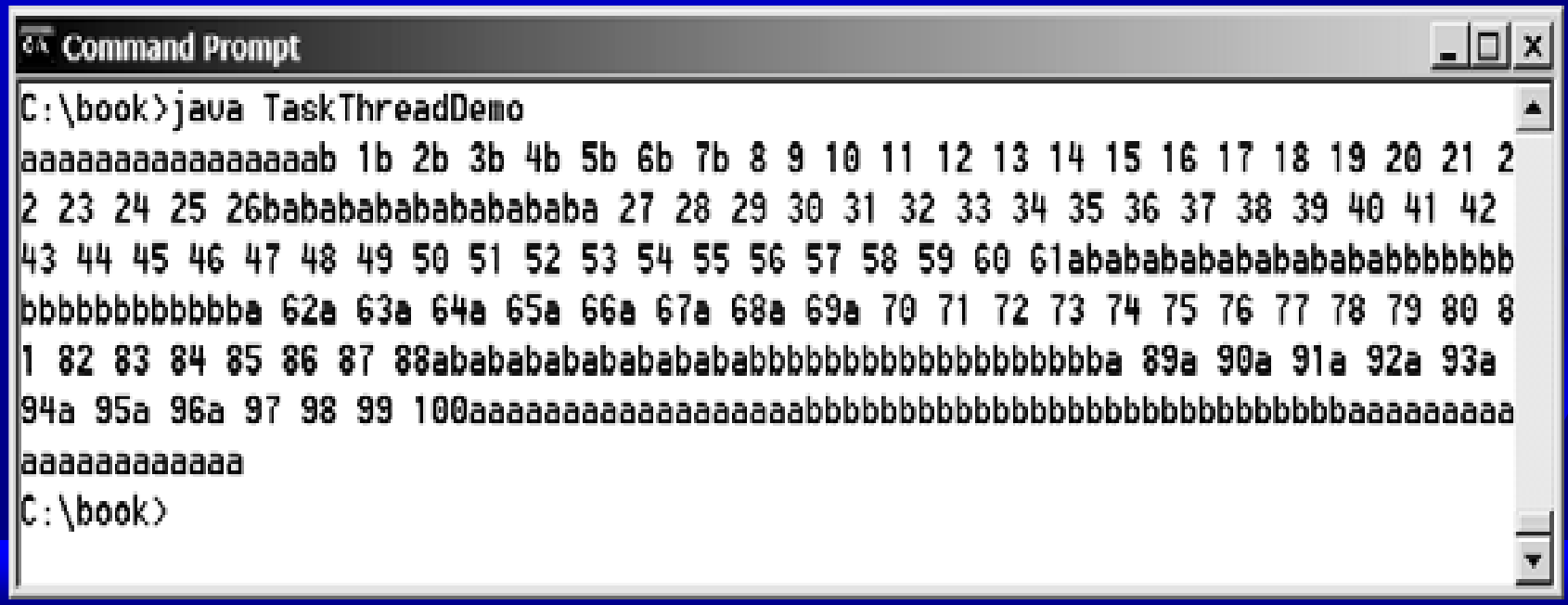

```
// Custom task class
public class TaskClass implements Runnable {
  ...
  public TaskClass(...) {
    ...
  }

  // Implement the run method in Runnable
  public void run() {
    // Tell system how to run custom thread
    ...
  }
  ...
}
```

```
// Client class
public class Client {
  ...
  public void someMethod() {
    ...
    // Create an instance of TaskClass
    TaskClass task = new TaskClass(...);

    // Create a thread
    Thread thread = new Thread(task);

    // Start a thread
    thread.start();
    ...
  }
  ...
}
```

Using the `Runnable` Interface to Create and Launch Threads

Example: Create and run three threads:

- The first thread prints the letter *a* 100 times.
- The second thread prints the letter *b* 100 times.
- The third thread prints the integers 1 through 100.



TaskThreadDemo.java

create tasks

create threads

start threads

task class

run

task class

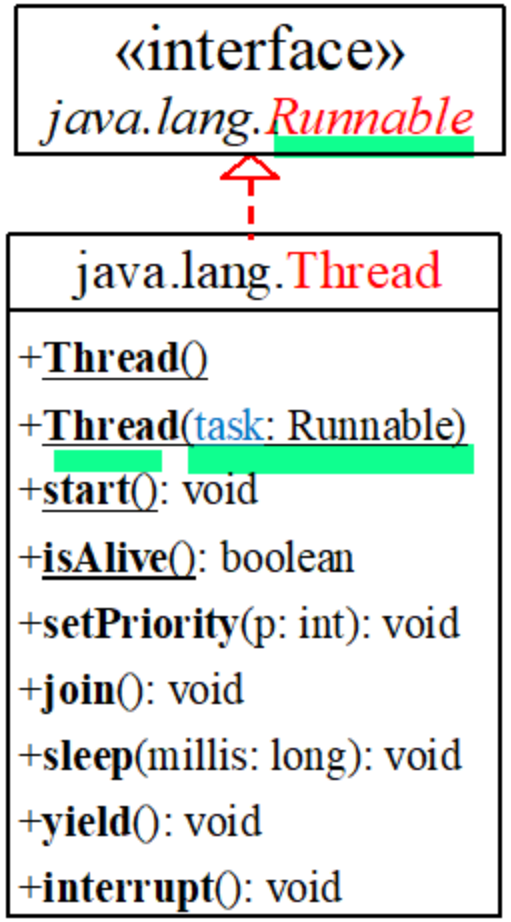
run

```
1  public class TaskThreadDemo {
2    public static void main(String[] args) {
3      // Create tasks
4      Runnable printA = new PrintChar('a', 100);
5      Runnable printB = new PrintChar('b', 100);
6      Runnable print100 = new PrintNum(100);
7
8      // Create threads
9      Thread thread1 = new Thread(printA);
10     Thread thread2 = new Thread(printB);
11     Thread thread3 = new Thread(print100);
12
13     // Start threads
14     thread1.start();
15     thread2.start();
16     thread3.start();
17   }
18 }
19
20 // The task for printing a character a specified number of times
21 class PrintChar implements Runnable {
22   private char charToPrint; // The character to print
23   private int times; // The number of times to repeat
24
25   /** Construct a task with specified character and number of
26    *  times to print the character
27    */
28   public PrintChar(char c, int t) {
29     charToPrint = c;
30     times = t;
31   }
32
33   /** Override the run() method to tell the system
34    *  what task to perform
35    */
36   public void run() {
37     for (int i = 0; i < times; i++) {
38       System.out.print(charToPrint);
39     }
40   }
41 }
42
43 // The task class for printing numbers fr
44 class PrintNum implements Runnable {
45   private int lastNum;
46
47   /** Construct a task for printing 1, 2,
48   public PrintNum(int n) {
49     lastNum = n;
50   }
51
52   /** Tell the thread how to run */
53   public void run() {
54     for (int i = 1; i <= lastNum; i++) {
55       System.out.print(" " + i);
56     }
57   }
58 }
```

Command Prompt

```
C:\book>java TaskThreadDemo
aaaaaaaaaaaaaaab 1b 2b 3b 4b 5b 6b 7b 8 9 10 11 12 13 14 15 16 17 18 19 20 21 2
2 23 24 25 26bababababababababa 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61abababababababababbbbbbb
bbbbbbbbbbbba 62a 63a 64a 65a 66a 67a 68a 69a 70 71 72 73 74 75 76 77 78 79 80 8
1 82 83 84 85 86 87 88abababababababababbbbbbbbbbbbbbbbbbbbbba 89a 90a 91a 92a 93a
94a 95a 96a 97 98 99 100aaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbaaaaaaaaa
aaaaaaaaaaaa
C:\book>
```

Thread Class



Creates a default thread.

Creates a thread for a specified task.

Starts the thread that causes the run() method to be invoked by the JVM.

Tests whether the thread is currently running.

Sets priority p (ranging from 1 to 10) for this thread.

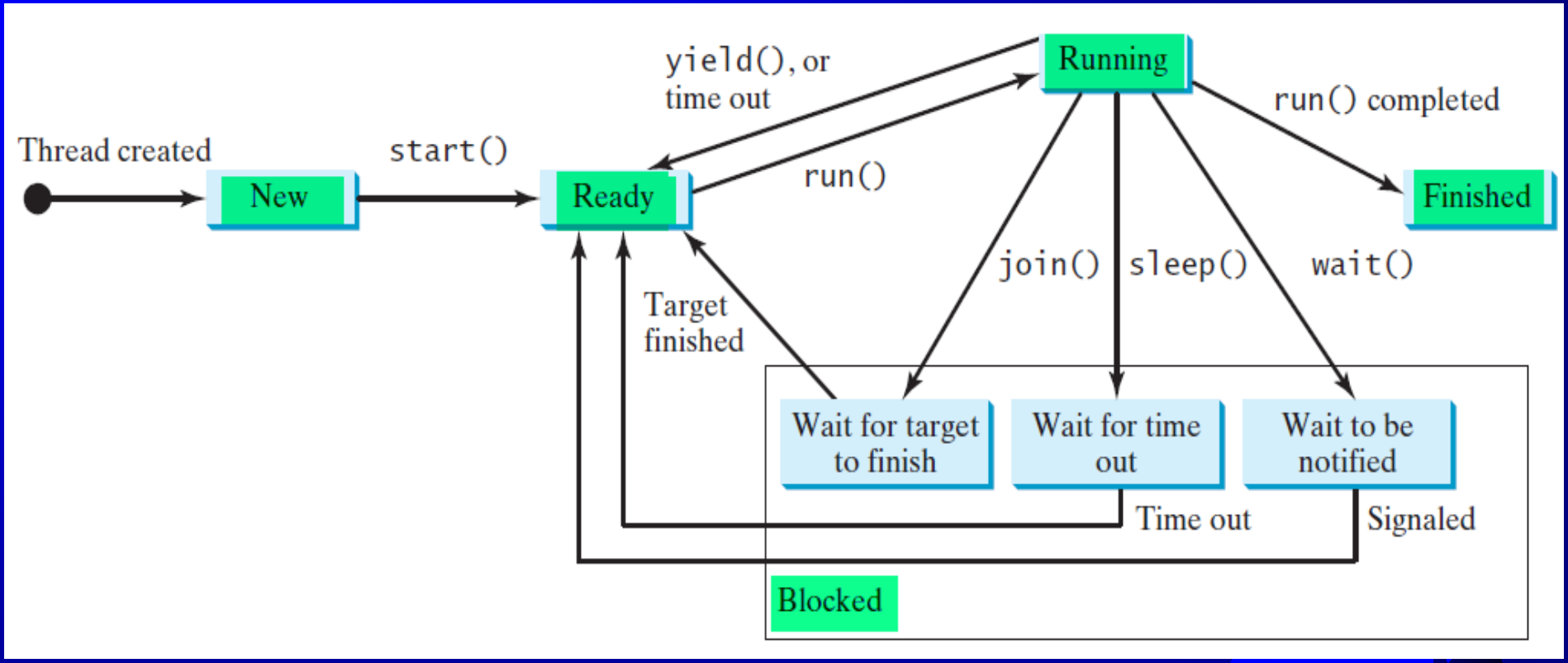
Waits for this thread to finish.

Puts the runnable object to sleep for a specified time in milliseconds.

Causes this thread to temporarily pause and allow other threads to execute.

Interrupts this thread.

Thread States



Threads can be in one of five states: New, Ready, Running, Blocked, or Finished .

Thread Priority

Java assigns every thread a priority.

- **Range: 1-10 （high）**
- Some constants：

```
Thread.MIN_PRIORITY : 1
Thread.NORM_PRIORITY: 5      （default priority）
Thread.MAX_PRIORITY : 10
```

By default, a thread's priority: its parent-thread' priority (that spawned it)

- the main thread's priority: Thread.NORM_PRIORITY.

Set/get the priority:

```
setPriority(int priority)
getPriority()
```

Thread Priority

The JVM always picks the currently runnable thread with the highest priority.

- A lowerpriority thread can run only when no higher-priority threads are running.
- If all runnable threads have equal priorities, each is assigned an equal portion of the CPU time in a circular queue. This is called the *round-robin scheduling*.

e.g., insert the following code at line 16 in previous example:

thread3.setPriority(Thread.MAX_PRIORITY);

- The thread3 will be finished first.

Static yield() Method

Temporarily release time for other threads.

e.g., modify Lines 53-57 as follows:

```
public void run() {
  for (int i = 1; i <= lastNum; i++) {
    System.out.print(" " + i);
    Thread.yield();
  }
}
```

Every time a number is printed, the print100 thread is yielded.

So, each number (print100 thread) is followed by some characters (printA/printB thread).

Static sleep(milliseconds) Method

Puts the thread to sleep for the specified time in milliseconds.

e.g., modify Lines 53-57 as follows:

```
public void run() {
  for (int i = 1; i <= lastNum; i++) {
    System.out.print(" " + i);
    try {
      if (i >= 50) Thread.sleep(1);
    }
    catch (InterruptedException ex) {
    }
  }
}
```

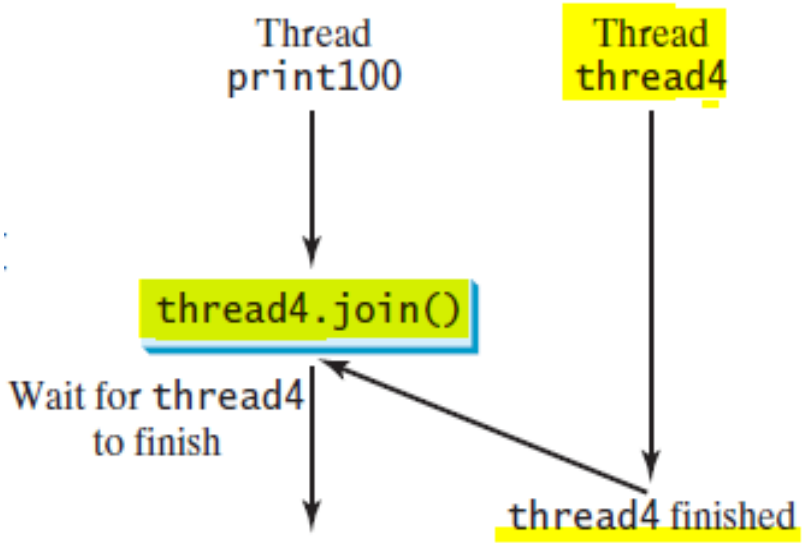
Every time a number (>= 50) is printed,
the print100 thread is put to sleep for 1 millisecond.

join() Method

Force one thread to wait for another thread to finish.

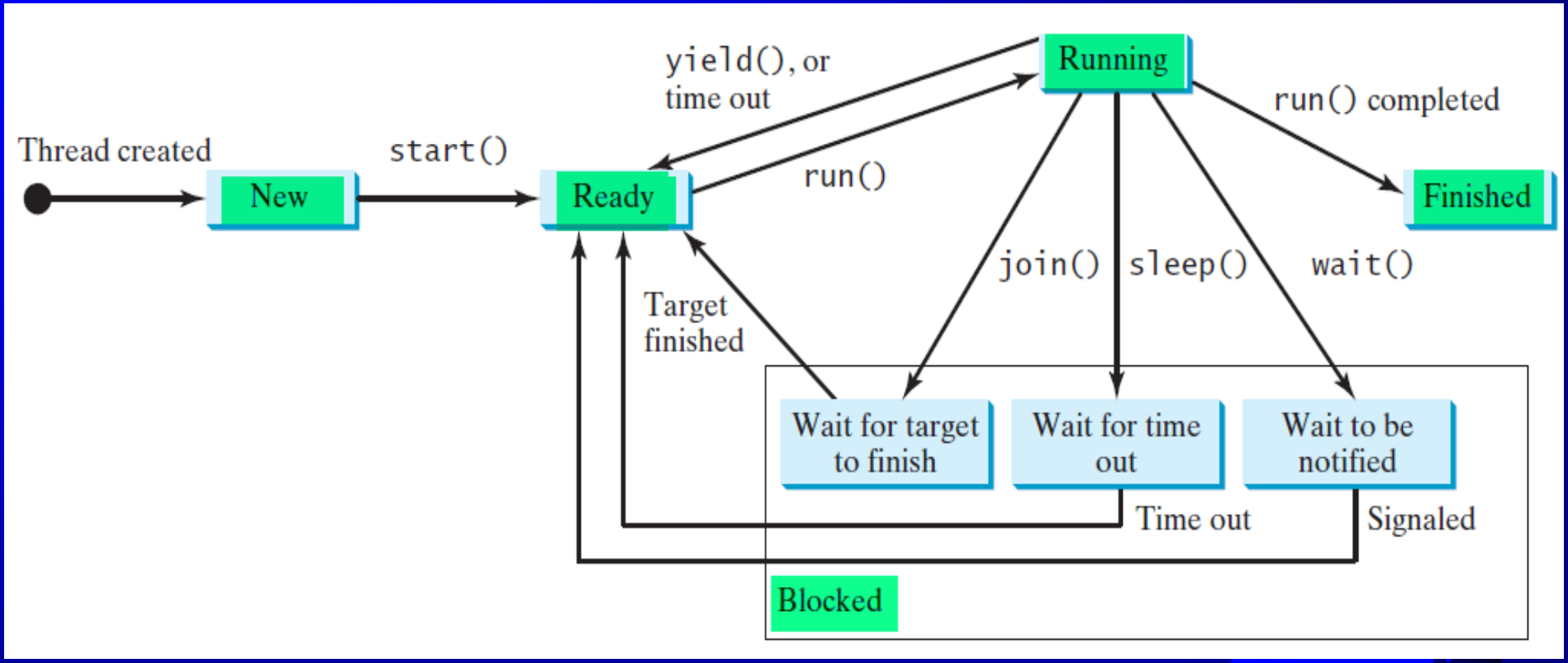
e.g., modify Lines 53-57 as follows:

```
public void run() {
  Thread thread4 = new Thread(
    new PrintChar('c', 40));
  thread4.start();
  try {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print(" " + i);
      if (i == 50) thread4.join();
    }
  }
  catch (InterruptedException ex) {
  }
```



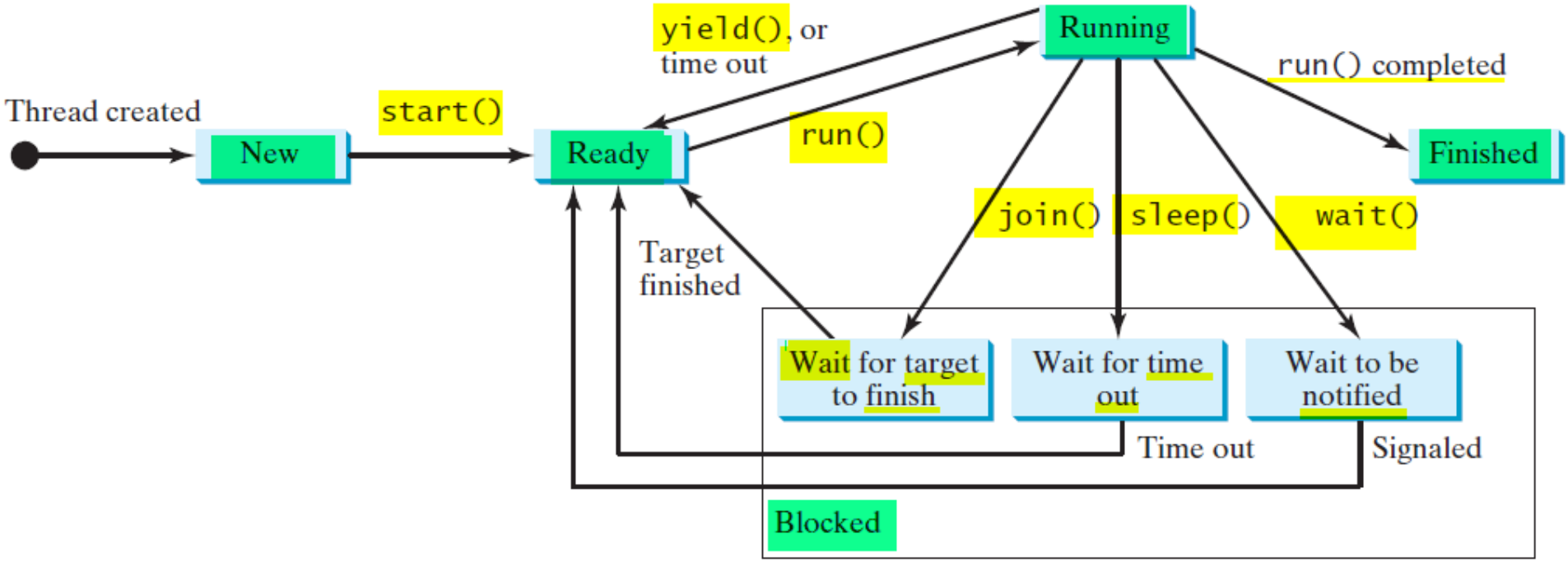
The numbers > 50 are printed after thread4 is finished.

Thread States



Threads can be in one of five states: New, Ready, Running, Blocked, or Finished

Thread States



Threads can be in one of five states: New, Ready, Running, Blocked, or Finished

interrupt(), isInterrupted()

interrupt() : interrupts a thread

If a thread is currently in the ***Ready/Running state***

- Set its interrupted flag ;

if a thread is currently ***Blocked***

- it is awakened and enters *Ready* state
- an java.io.InterruptedException is thrown.

isInterrupt() : tests whether the thread is interrupted.

isAlive()

isAlive() :

true: a thread is in *Ready* /*Running* /*Blocked* state;
false: *New&NotStarted* / *Finished*.

To **stop/finish a thread :**

Assign *null* **to** *a Thread variable*

– rather than use the stop() method : outdated

FlashText.java

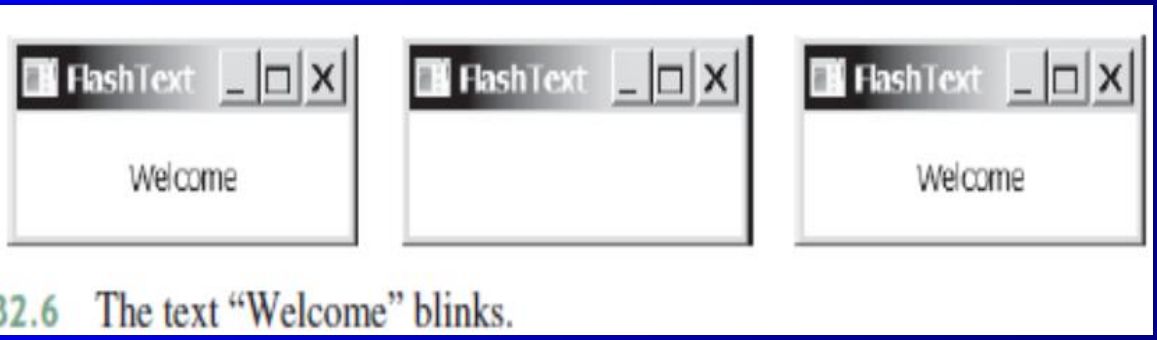


FIGURE 32.6 The text "Welcome" blinks.

```
17      new Thread(new Runnable() {                                    create a thread
18        @Override
19        public void run() {                                          run thread
20          try {
21            while (true) {
22              if (lblText.getText().trim().length() == 0)            change text
23                text = "Welcome";
24              else
25                text = "";
26
27              Platform.runLater(new Runnable() { // Run from JavaFX GUI   Platform.runLater
28                @Override
29                public void run() {
30                  lblText.setText(text);                             update GUI
31                }
32              });
33
34              Thread.sleep(200);                                     sleep
35            }
36          }
37          catch (InterruptedException ex) {
38          }
39        }
40      }).start();
```

FlashText.java

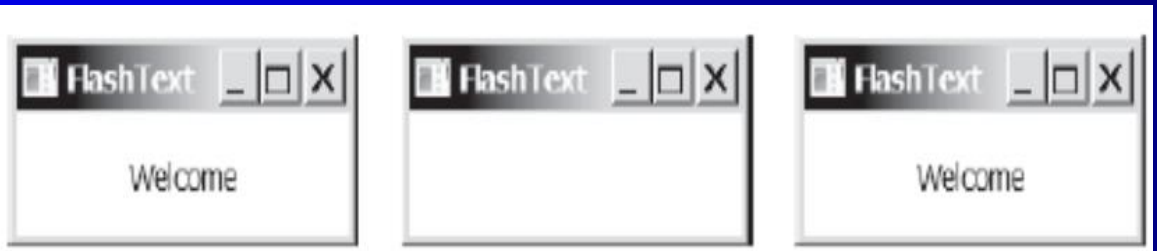


FIGURE 32.6 The text "Welcome" blinks.

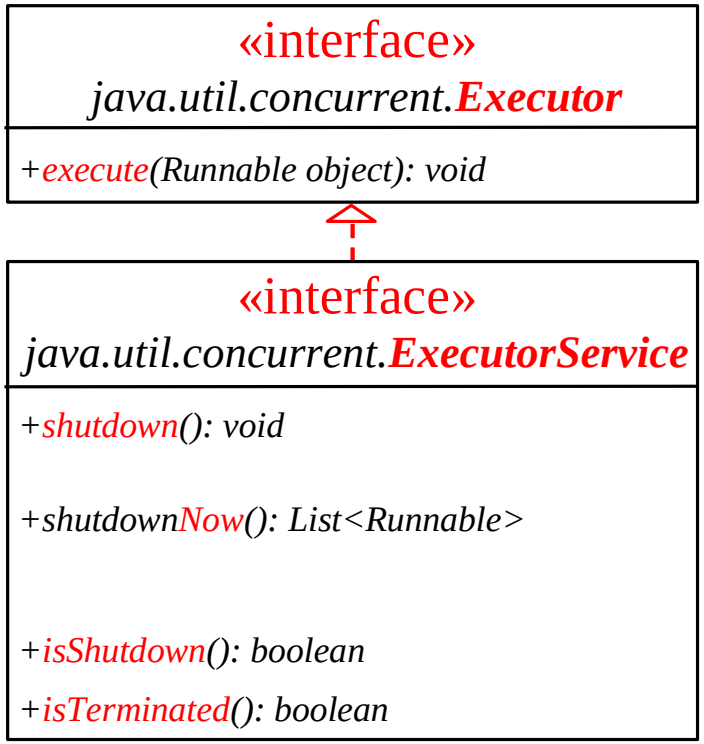
```
17      new Thread(new Runnable() {                                    create a thread
18        @Override
19        public void run() {                                          run thread
20          try {
21            while (true) {
22              if (lblText.getText().trim().length() == 0)            change text
23                text = "Welcome";
24              else
25                text = "";
26
27              Platform.runLater(new Runnable() { // Run from JavaFX GUI   Platform.runLater
28                @Override
29                public void run() {
30                  lblText.setText(text);                             update GUI
31                }
32              });
33
```

JavaFX GUI is run from the *JavaFX application thread*. The flashing control is run from a separate thread. The code in a nonapplication thread cannot update GUI in the application thread. To update the text in the label, a new `Runnable` object is created in lines 27–32. Invoking `Platform.runLater(Runnable r)` tells the system to run this `Runnable` object in the application thread.

Thread Pools

A **Thread Pool** : manage the tasks executing concurrently.

- Starting a new thread for each task could cause poor performance.
- **Executor interface** for executing tasks in a thread pool
  ExecutorService interface for managing tasks.
  - ExecutorService is a subinterface of Executor.



Executes the runnable task.

Shuts down the executor, but allows the tasks in the executor to complete. Once shutdown, it cannot accept new tasks.

Shuts down the executor immediately even though there are unfinished threads in the pool. Returns a list of unfinished tasks.

Returns true if the executor has been shutdown.

Returns true if all tasks in the pool are terminated.

Creating Executors

To create an Executor object,

use the static methods in the **Executors class**:

java.util.concurrent.**Executors**	
+newFixedThreadPool(numberOfThreads: int): ExecutorService	Creates a thread pool with a fixed number of threads executing concurrently. A thread may be reused to execute another task after its current task is finished.
+newCachedThreadPool(): ExecutorService	Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.

Thread Synchronization

A **shared resource** may be corrupted if it is accessed simultaneously by multiple threads.

- e.g., two unsynchronized threads accessing the same bank account (shared data member ".balance") may cause conflict.
- two threads: each adds a penny to an account; the account is initially 0.
- thread[i] did nothing: results are overwritten

Step	balance	thread[i]	thread[j]
1	0	newBalance = bank.getBalance() + 1;	
2	0		newBalance = bank.getBalance() + 1;
3	1	bank.setBalance(newBalance);	
4	1		bank.setBalance(newBalance);

Race Condition

Race condition among multi-threads: multi-threads are accessing a shared resource in a way that causes conflict.

Lead to : ***Not thread-safe class***

A **class** is ***thread-safe*** *:* if an object of the class does not cause a race condition in the presence of multiple threads.

`synchronized` method

To avoid race conditions:

Prevent more than one thread from simultaneously entering certain part of the program, known as critical region.

Use the **synchronized** keyword to synchronize the method so that only one thread can access the method at a time.

public **synchronized** void deposit(double amount)

synchronized Block /Statements

Synchronized block : synchronized statements

```
synchronized (expr) {
   statements;
}
```

- `expr`: *an **object** reference.*

- If the object is already locked by another thread, the thread is blocked.
- If a lock is obtained on the object, the synchronized block is executed, and then the lock is released.

Synchronizing block vs. Method

a **synchronized method:**

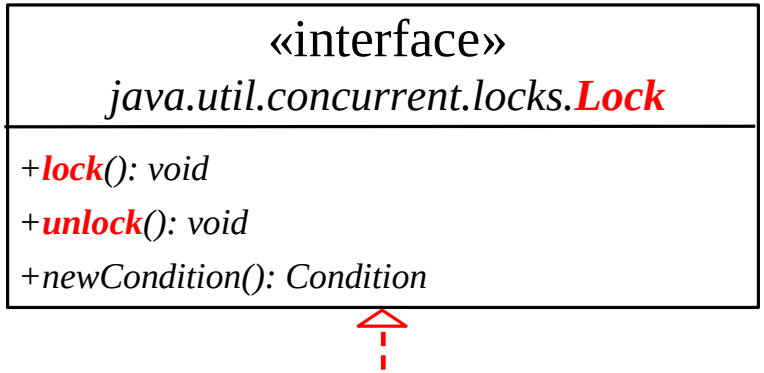
```
public synchronized void xMethod() {
   // method body
}
```

equivalent to **a synchronized block:**

```
public void xMethod() {
   synchronized (this) {
      // method body
   }
}
```

Synchronization Using Locks

A lock is an instance of the **Lock interface**, which declares the methods for acquiring and releasing locks.



Acquires the lock.

Releases the lock.

Returns a new Condition instance that is bound to this Lock instance.

Same as ReentrantLock(false).

Creates a lock with the given fairness policy. When the fairness is true, the longest-waiting thread will get the lock. Otherwise, there is no particular access order.

Thread Cooperation

A thread can specify what to do under a certain condition.

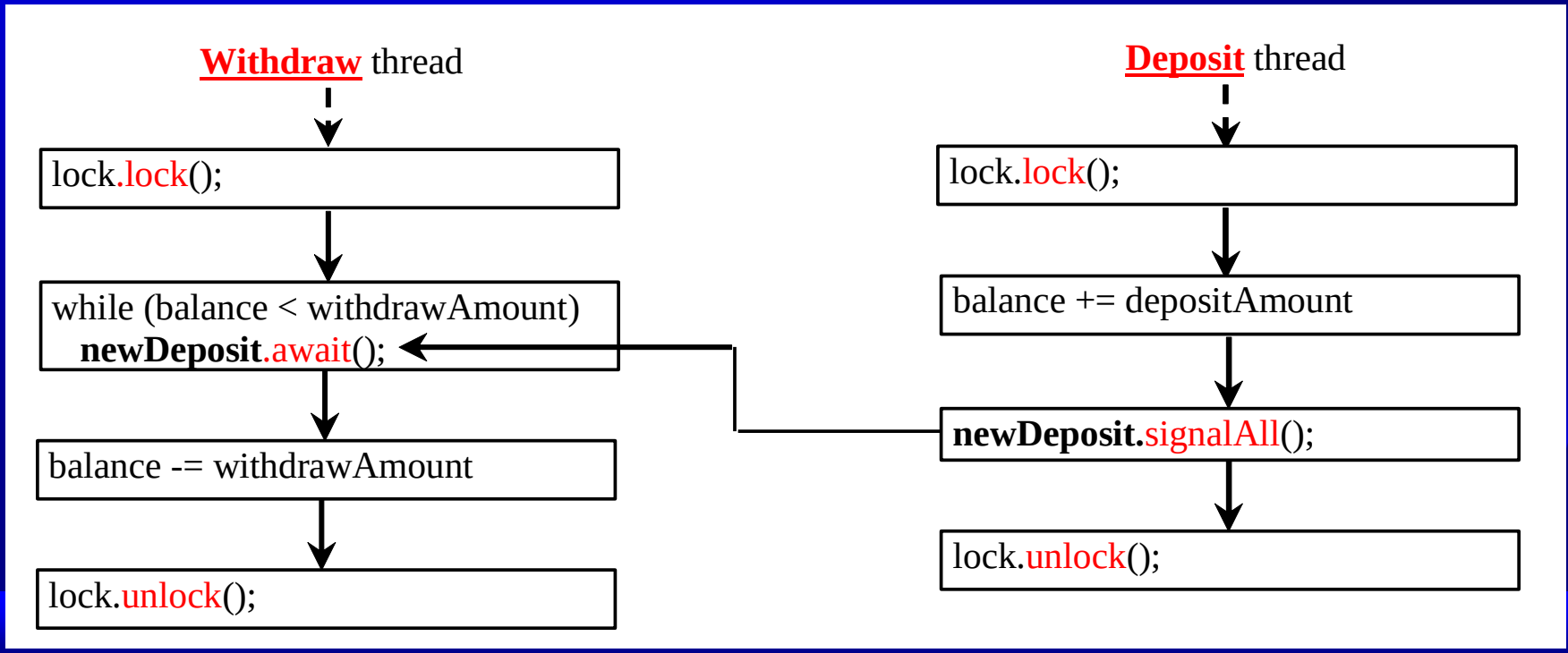
Condition:
- object created by invoking **newCondition()** on a Lock object.
- methods:
  await(), signal(), signalAll()

«interface» *java.util.concurrent.**Condition***	
+*await(): void*	Causes the current thread to wait until the condition is signaled.
+*signal(): void*	Wakes up one waiting thread.
+*signalAll(): Condition*	Wakes up all waiting threads.

Example: Thread Cooperation

Two cooperated threads: Deposit & Withdraw ,
use a lock with a condition: **newDeposit**

- Withdraw thread : wait for the newDeposit condition
  if the balance is less than the amount to be withdrawn
- Deposit thread: signals the waiting Withdraw thread to try again
  when adding money to the account



Example: Thread Cooperation

two threads: one deposits to an account, one withdraws from it.

- the initial balance is 0
- the amount to deposit/withdraw is random.