Concurrency

1. Why concurrent programming?

- Performance
- Throughput
- Utilization of system resources

2. Evolution

- **Single user system**:
    - First systems supported one single activity at a time.
    - Late 1950s - One general-purpose processor and one or more special-purpose processors for input and output operations

- **Multiprocessing systems**: These systems use the CPU while an input/output operation is being performed. Note that programmers cannot control this task explicitly.

- **Multitasking systems**: These systems give the impression that there are several CPUs serving different users at the same time. Programmers cannot control task scheduling. This is done by the operating system.

- **Multithreading systems**: This is becoming very popular with Java. Programmers can split their programs into several threads and schedule them.

- **Multiprocessor systems**:

    - These are systems with several processing elements (PEs) and programmers can concurrently use all these PEs.
    - Single-Instruction Multiple-Data (SIMD) machines:
        - The same instruction goes to all processors, each with different data - e.g., *vector processors*
        - Multiple-Instruction Multiple-Data (MIMD) machines: Independent processors that can be synchronized (unit-level concurrency)

3. Definitions

- **Concurrency or Parallelism?**

- ***Concurrency***:
  - Logically simultaneous processing.
  - Does not require multiple processing elements
  - Requires interleaved execution on a single processing element.

- ***Parallelism***:

  - Physically simultaneous processing.

  - It does involve several processing elements.

- Both concurrency and parallelism require controlled access to shared resources.
- In general people use the word concurrent and parallel interchangeably.

- **A concurrent program**: It is a program that has multiple threads or tasks of control allowing it perform multiple computations in parallel and to control multiple external activities that occur at the same time.

- **Processes:**
  - A process is an operating system abstraction that allows once computer system to support many units of execution.
  - Each process typically represents a separate running program; for example, a web browser.

- A process can generally be composed of one or several threads.

- **Threads**:
  - A thread is a single sequential execution path in a program.
  - A thread is executed independently of other threads, while at the same time possibly sharing underlying system resources such as files, as well as accessing other objects constructed within the same program.
  - Every program has at least one thread
  - Threads subdivide the run-time behavior of a program into separate, independently running subtasks.
  - Every thread has its own:
    - stack,
    - priority, and
    - virtual set of registers.

4. Concurrent languages

- Concurrent Pascal
- Concurrent C
- Communicating Sequential Processes (CSP)
- Ada
- Java
- Etc.

5. Problems with concurrency

- **Non-deterministic**: Unlike sequential programs, where programs are completely deterministic and their behavior can be reproducible, concurrent programs are likely to be highly non-deterministic. The order of execution of process in a concurrent program is unpredictable since it may be influenced by run-time conditions.
- **Speed-dependence:** A sequential program is *speed-independent* because its correctness does not depend on the rate at which it is executed. However, a concurrent program may be *speed-dependent*. Its final output may depend on the relative speeds of execution of its component sequential processes.
- **Deadlock**: Deadlock is a situation in which a set of processes are prevented from making any further progress by their mutually incompatible demands for additional resources. This can occur in a system of processes and resources iff the following conditions all hold together:
    - ✓ **Mutual exclusion**: processes are given exclusive access to the resources they acquire.
    - ✓ **Wait and hold**: processes continue to hold previously allocated resources while waiting for a new resource demand to be satisfied.

- **No preemption**: resources cannot be removed from a process until it voluntarily releases them.
- **Circular wait:** there may be a cycle of resources and processes in which each process is awaiting resources that are held by the next process in the cycle.

- **Starvation**: This is the case where a process is prevented indefinitely from running by unfair scheduling. Fair scheduling ensures that no process needing a resource is indefinitely prevented from obtaining it by the demand from other processes.

6. Process Interactions

- **Definitions and Notations [D. Watt]:**

  - **Sequential processes**:
    - Given two processes A and B, the sequential execution of A and B is denoted **A;B,** i.e., A is executed before B.

- **Collateral processes**:
  - Given two processes A and B, the collateral execution of A and B is denoted **A,B**, i.e., the execution of A and B can be done in any order.
  - Example: m=7, n=n+1
  - Note that collateral processes are non-deterministic.
- **Parallel processes**:
  - Given two processes A and B, the parallel execution of A and B is denoted **A||B**.
  - Note that A and B don't have to be executed simultaneously.
  - Unlike sequential processes, A and B may need to interact.

- **Independent processes**

  - **Definition**: Two processes A and B are independents if any component or any task A_i of A may be executed in any time relationship to any component Bi of B, without effect on the meaning of the program.
  - Note that if A and B are independent, it follows that A;B is equivalent to B;A
  - Also we have A,B is equivalent to A||B.
  - In general, it is undecidable whether A and B are independent.

- **Competing processes**

- ✓ Definition: Two processes A and B compete if each must gain exclusive access to the same resource R for some of their tasks.
- ✓ Let us assume the following:
  - A be the sequence A1, A2, A3
  - B be the sequence B1, B2, B3
  - None of A1, A3, B1, B3 uses R
  - We also assume that A1 and B1are independent, and that A3 and B3 are independent.
  - A2 and B2 want to access the same resource R.

  - ▪ A2 and B2 must not take place simultaneously.
  - ▪ They are called ***critical sections*** with respect to the resource R.
  - ▪ Thus the execution of the two sequence of command A and B will have two possible outcome:

...; A2; ...; B2;...

or

...; B2; ...; A2;...

but not

...; A2 || B2;...

Which of these outcomes actually happens will depend on the relative speeds at which A and B are executed, and is not in general predictable.

- **Communicating processes**

  - ✓ **Definition**: We say that there is a communication from A to B if the task A2

must entirely precede the action B2 in the case B2 needs information produced by A2.

- ✓ Two communicating concurrent processes **A||B** have the same outcome as the sequential execution of **A;B**.
- ✓ Note that communicating processes involving several processes yield to the **pipeline** paradigm.
- ✓ **Example**: In **Unix** provides the notation C1|C2 to execute two commands C1 and C2 concurrently and C2 takes the output of C1 as input.
- ✓ Two processes **intercommunicate** if there is a communication in both directions.
- ✓ This makes the possible outcomes of 'C || k' **very much more numerous**.

7. Low-level Concurrency Primitives

- What are low-level abstractions that affect concurrency: create, destroy, and control.

- **Process creation and control**:
  - ✓ Different programming languages provide different primitive operations to create and control processes:

    Fork: create, load, and start a task
    Join: wait and destroy

- In Java, this is done by creating an object of type thread and start it by invoking the start method.

  Example:

  New MyThread().start();

  where MyThread is a class thread.

  By invoking the start() method, the JVM scheduler is told to run the thread. The scheduler invokes the run() method from the object.

  Threads in Java are destroyed once the run() method is done much like the main() function in a C program.

- **Event**

  - An event represents a class of state changes, the occurrence of which must be communicated among a set of processes.

  - In general, this is done through two operations:

    Event-wait(e): When a process executes this event, it is blocked awaiting the next occurrence of an event e.
    Event-signal(e): This operation makes all processes that are waiting for the event e to run again. If e is omitted, then all blocked processes are ready to run.

- Java uses the following operations:

  Wait()
  Notify() or notifyall()

- **Messages**:
  - When processes run on a network of computers, the network provides a data communication service that supports process interaction by the exchange of messages.
  - If the set of computer uses a shared memory architecture, the communication can be done using shared variable construct such as in Universal Parallel C (UPC).

- **Remote Procedure Calls (RPCs)**

  - RPC is based on **extending the notion of conventional, or local procedure calling**, so that the called procedure need not exist in the same address space as the calling procedure.

- The two processes may be on the same system, or they may be on different systems with a network connecting them. By using RPC, programmers of distributed applications avoid the details of the interface with the network.
- The transport independence of RPC isolates the application from the physical and logical elements of the data communications.
- **Remote Procedure Calling Mechanism** A remote procedure is uniquely identified by the triple: (program number, version number, procedure number):
    - The **program number** identifies a group of related remote procedures, each of which has a unique procedure number.
    - A program may consist of one or more versions. Each version consists of a collection of procedures that are available to be called remotely. Version numbers enable multiple versions of an RPC protocol to be available simultaneously.
    - Each procedure has a procedure number.

8. Synchronization

- Data corruption and inconsistency, and deadlock may occur in a multiprogramming environment may occur when processes share resources like the content of a shared memory area, files, devices.
- In order to control the access to these shared resources, multithreaded environments should include support for concurrency control mechanisms.
- Three type to handle synchronization:

  - ✓ **Semaphores**
  - ✓ **Conditional critical sections**: This is similar to the critical section, except that the execution of the critical section is based on a condition(s)
  - ✓ **Monitors**: allows safe and effective sharing of an ADT among several processes.
  - ✓ **Message Passing**

- Semaphores

  - Dijkstra – 1965
  - A *semaphore* is a data structure consisting of a counter and a queue for storing task descriptors
  - Semaphores can be used to implement guards on the code that accesses shared data structures.

- Semaphores have only two operations, wait and release (originally called P and V by Dijkstra)

  P(s): if s > 0 then s = s - 1
  else suspend current process
  V(s): if there is a process suspended on s
  then wake one of them up
  else s = s + 1

- Semaphores can be used to provide both competition and cooperation synchronization

- Monitor

  - The idea: encapsulate the shared data and its operations to restrict accessA *monitor* is an abstract data type for shared data

  - It is the ***implementation*** that ensures the mutual exclusion using the **delay** and **continue** primitives (p 223 & 225)

  - *Example language:* Concurrent Pascal and Java

- Message Passing

  - Message passing is a general model for concurrency
  - It is not just for competition synchronization

- *Central idea:* task communication is like seeing a doctor--most of the time he waits for you or you wait for him, but when you are both ready, you get together, or rendezvous (don't let tasks interrupt each other)
- In terms of tasks, we need:
    - A mechanism to allow a task to indicate when it is willing to accept messages
    - Tasks need a way to remember who is waiting to have its message accepted and some "fair" way of choosing the next message

- Definition: When a sender task's message is accepted by a receiver task, the actual message transmission is called a *rendezvous*

- Example: The Ada Message-Passing Model

9. Example

- The classic *Producer-Consumer* problems
    - Producer creates things
    - Consumer processes things
    - Many examples in operating systems
        - ✓ Print spooler
        - ✓ Compile sequences (Compile, generate code, link, etc.)

- Pseudo Code:

```
Process Producer  {
  repeat forever  {
    produce an element;
    append it to the buffer;
  }
}
Process Consumer  {
 repeat forever  {
    remove an element from the buffer;
    operate on it;
  }
}
```

- We need to control access to the buffer

  - ✓ Writing to a full buffer
  - ✓ Reading from an empty buffer
  - ✓ More subtly, we need to ensure ***mutual exclusion*** to the variable describing the state of the buffer

```
Procedure Append  {
  total_elements++;
  i = next_in( );
  buffer[i] = x;
}
Procedure Remove  {
  total_elements--;
  j = next_out( );
  x = buffer[j];
}
```

- Producer-Consumer with semaphores
  - These are the variables shared between the producer and the consumer:

```
// Buffer to hold items.
buffer buf;
semaphore mutex = 1;          // Mutal exclusion.
          items = 0;          // Num. items in buf.
          spaces = buf.size(); // Num. spaces in buf.
```

  - Producer

```
process Producer  {
  int i;
  for( ;; )  {
    produce( i );
    P( spaces ); // Wait for free space.
    P( mutex );  // Wait to access buf.
    buf.append( i );
    V( mutex );  // Release buf for others.
    V( items );  // One more item in buf.
  }
}
```

  - Consumer

```
process Consumer  {
  int j;
  for( ;; )  {
    P( items );  // Wait for item in buf.
    P( mutex );  // Wait to access buf.
    j = buf.remove( );
    V( mutex );  // Release buf for others.
    V( spaces ); // One more space in buf.
```

```
    consume( j );
  }
}
```

10. Concurrency in Java

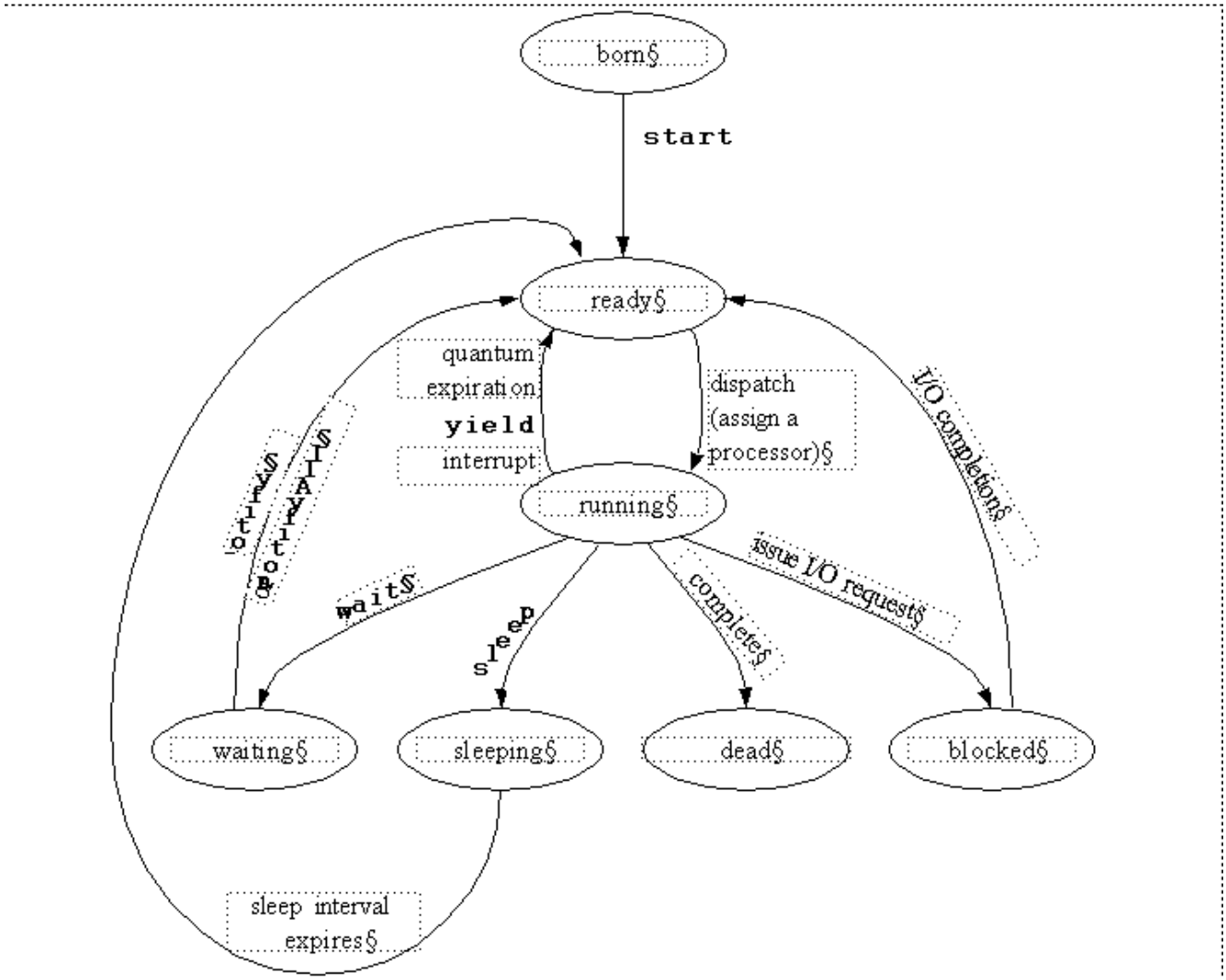
- **Introduction**

    - Java provides a **Thread** class to support threading.
    - This class keeps track of the context of threads and provides many methods to control them.
    - Java provides a package, called **green threads**, to support operating systems that do not support threads.

- **Creation of a Java thread**

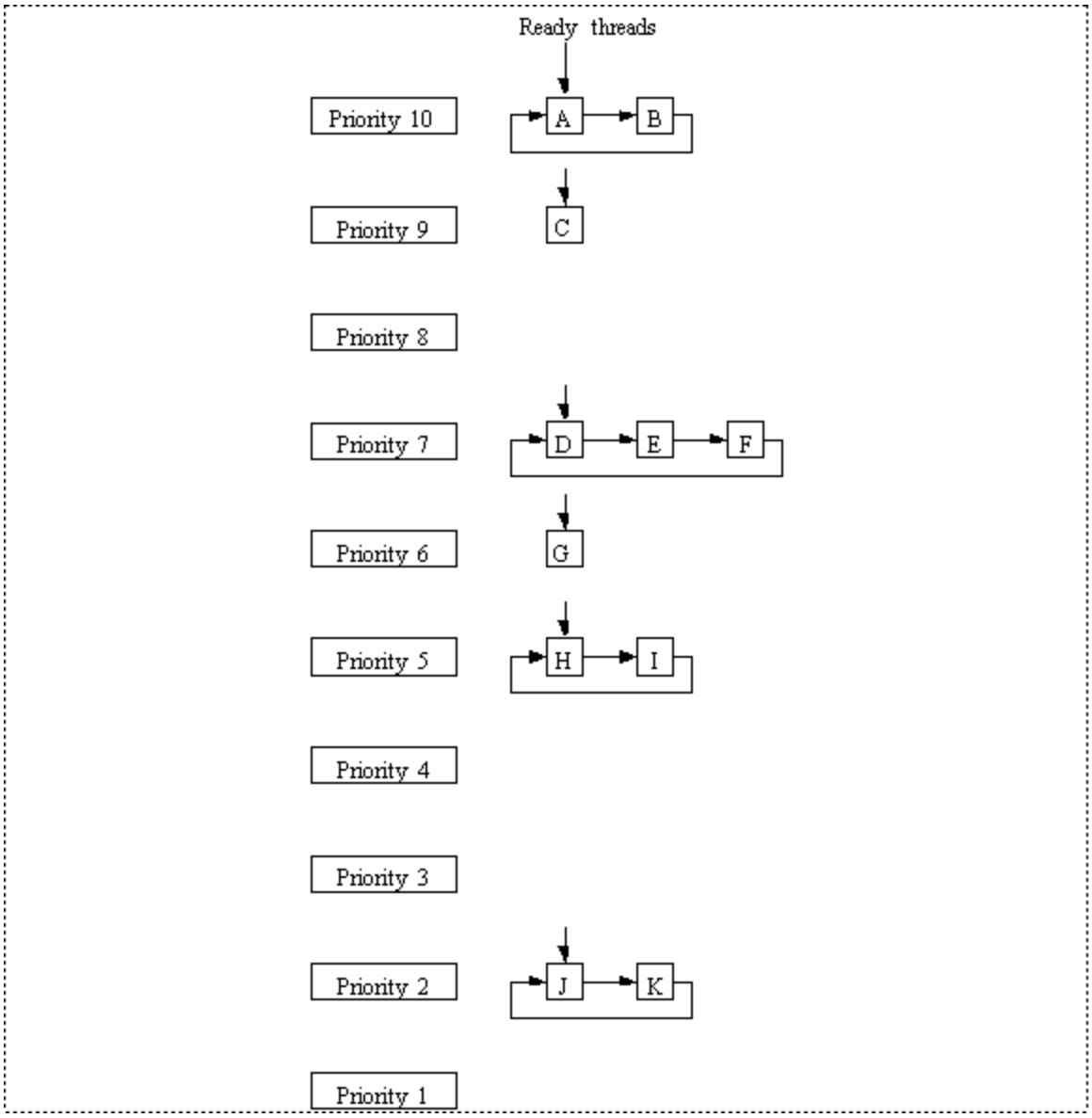
1. Create a new class:
    a. Define a subclass of Thread
    b. Override its run() method
2. Instantiate and run the thread
    a. Create an instance of the class
    b. Call its start() method, which puts the thread in the queue.
3. The JVM scheduler calls the thread's run() method.

11. Java Thread Life Cycle (Deitel & Deitel)



↳ Thread priority:

- Every Java thread has a priority in the range **Thread.MIN_PRIORITY** (a constant of 1) and **Thread.MAX_PRIORITY** (a constant of 10).
- Default: Each thread is given priority **Thread.NORM_PRIORITY** (a constant of 5).
- Each new thread inherits the priority of the thread that creates it.
- The job of the Java *scheduler* is to keep a highest-priority thread running at all times, and if timeslicing is available, to ensure that several equally high-priority threads each execute for a quantum in *round-robin* fashion (i.e., these threads can be timesliced).
- The following figure illustrates Java's multilevel priority queue for threads. In the figure, threads A and B each execute for a quantum in round-robin fashion until both threads complete execution. Next, thread C runs to completion. Then, threads D, E and F each execute for a quantum in round-robin fashion until they all complete execution. This process continues until all threads run to completion.



- **Starvation:** Note that new higher-priority threads could postpone–possibly indefinitely–the execution of lower-priority threads. Such *indefinite postponement* is often referred to more colorfully as *starvation*.

↳ Example:

```
package paradigms;

 1) // Fig. 15.3: ThreadTester.java
 2) // Show multiple threads printing at different intervals.
 3)
 4) public class ThreadTester {
 5)    public static void main( String args[] )
 6)    {
 7)       PrintThread thread1, thread2, thread3, thread4;
 8)
 9)       thread1 = new PrintThread( "thread1" );
10)       thread2 = new PrintThread( "thread2" );
11)       thread3 = new PrintThread( "thread3" );
12)       thread4 = new PrintThread( "thread4" );
13)
14)       System.out.println( "\nStarting threads" );
15)
16)       thread1.start();
17)       thread2.start();
18)       thread3.start();
19)       thread4.start();
20)
21)       System.out.println( "Threads started\n" );
22)    }
23) }
24)
25) class PrintThread extends Thread {
26)    private int sleepTime;
27)
28)    // PrintThread constructor assigns name to thread
29)    // by calling Thread constructor
30)    public PrintThread( String name )
31)    {
32)       super( name );
33)
34)       // sleep between 0 and 5 seconds
35)       sleepTime = (int) ( Math.random() * 5000 );
36)
37)       System.out.println( "Name: " + getName() +
38)                           ";  sleep: " + sleepTime );
39)    }
40)
41)    // execute the thread
42)    public void run()
```

```
43)    {
44)       // put thread to sleep for a random interval
45)       try {
46)          System.out.println( getName() + " going to sleep" );
47)          Thread.sleep( sleepTime );
48)       }
49)       catch ( InterruptedException exception ) {
50)          System.out.println( exception.toString() );
51)       }
52)
53)       // print thread name
54)       System.out.println( getName() + " done sleeping" );
55)    }
56) }
```

- The **PrintThread** constructor (line 30) initializes **sleepTime** to a random integer between 0 and 4999 (0 to 4.999 seconds). Then, the name of the thread and the value of **sleepTime** are output to show the values for the particular **PrintThread** being constructed.
- The name of each thread is specified as a **String** argument to the **PrintThread** constructor and is passed to the superclass constructor at line 32.
- *Note:* It is possible to allow class **Thread** to choose a name for your thread by using the **Thread** class's default constructor.
- When a **PrintThread**'s **start** method (inherited from **Thread**) is invoked, the **PrintThread** object enters the *ready* state.
- When the system assigns a processor to the **PrintThread** object, it enters the *running* state and its **run** method begins execution.
- Method **run** prints a **String** in the command window indicating that the thread is going to sleep then invokes the **sleep** method (line 47) to immediately put the thread into a *sleeping* state.

- When the thread awakens, it is placed into a *ready* state again until it is assigned a processor.
- When the **PrintThread** object enters the *running* state again, it outputs its name (indicating that the thread is done sleeping), its **run** method terminates and the thread object enters the *dead* state.
- Class **ThreadTester**'s **main** method (line 5) instantiates four **PrintThread** objects and invokes the **Thread** class **start** method on each one to place all four **PrintThread** objects in a *ready* state.
- Note that the program terminates execution when the last **PrintThread** awakens and prints its name. Also note that the **main** method terminates after starting the four **PrintThread**s, but the application does not terminate until the last thread dies.

🡻 Synchronization

- ✓ Java uses ***monitors*** to perform synchronization.
- ✓ Every object with **synchronized** methods is a monitor.
- ✓ The monitor allows one thread at a time to execute a **synchronized** method on the object.
- ✓ This is accomplished by *locking* the object when the **synchronized** method is invoked– also known as *obtaining the lock*.
- ✓ If there are several **synchronized** methods, only one **synchronized** method may be active on an object at once; all other threads attempting to invoke **synchronized** methods must wait.

- A thread executing in a **synchronized** method may determine that it cannot proceed, so the thread ***voluntarily*** calls **wait**.
- When a **synchronized** method finishes executing, the lock on the object is released and the monitor lets the highest-priority *ready* thread attempting to invoke a **synchronized** method proceed.