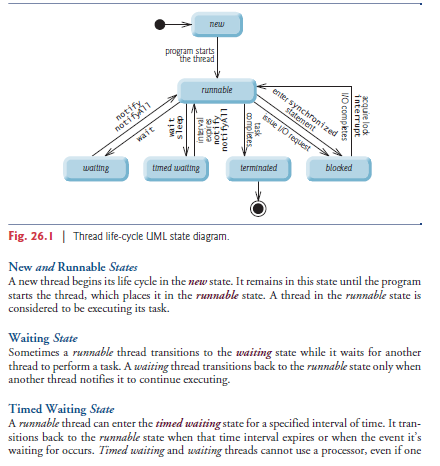
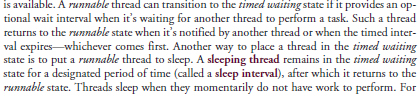
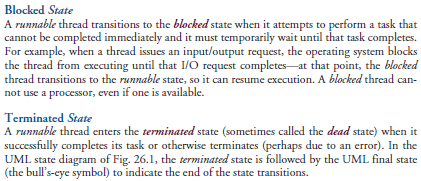
**MUTLI- THREADING**

Java makes concurrency available to you through the language and APIs. Java programs can have multiple **threads of execution**, where each thread has its own method-call stack and program counter, allowing it to execute concurrently with other threads while sharing with them application-wide resources such as memory. This capability is called **multithreading**.

Synchronized thread basically means that the threads actions are coordinated. That is one works in collaboration with other threads for smooth functioning of the system. The Java Virtual Machine (JVM) creates threads to run programs and threads to perform housekeeping tasks such as garbage collection.

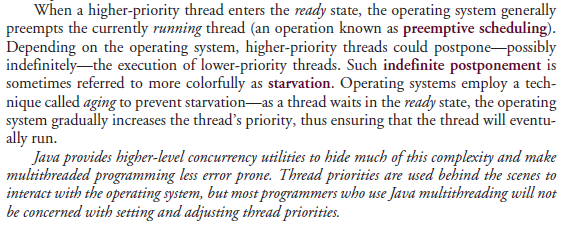






Every Java thread has a **thread priority** that helps determine the order in which threads are scheduled. Each new thread inherits the priority of the thread that created it. Informally, higher-priority threads are more important to a program and should be allocated processor time before lower-priority threads. Nevertheless, thread priorities cannot guarantee the order in which threads execute.

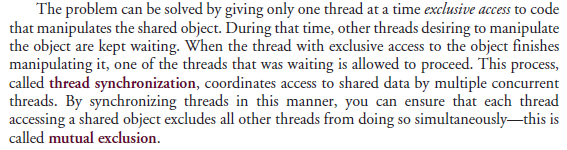
Most operating systems support timeslicing, which enables threads of equal priority to share a processor. An *operating system’s* **thread scheduler** determines which thread runs next. One simple thread-scheduler implementation keeps the highest-priority thread *running* at all times and, if there’s more than one highest-priority thread, ensures that all such threads execute for a quantum each in **round-robin** fashion. This process continues until all threads run to completion.

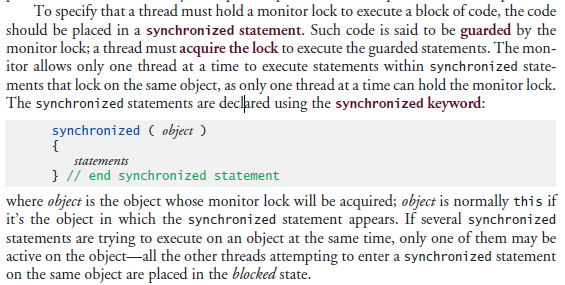


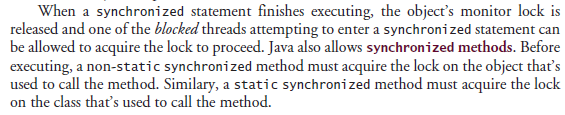
You implement the **Runnable** interface (of package java.lang) to specify a task that can execute concurrently with other tasks. The Runnable interface declares the single method **run**, which contains the code that defines the task that a Runnable object should perform. To allow a Runnable to perform its task, you must execute it. An **Executor** object executes Runnables. An Executor does this by creating and managing a group of threads called a **thread pool**. When an Executor begins executing a Runnable, the Executor calls the Runnable object’s run method, which executes in the new thread. The Executor interface declares a single method named **execute** which accepts a Runnable as an argument. The Executor assigns every Runnable passed to its execute method to one of the available threads in the thread pool. If there are no available threads, the Executor creates a new thread or waits for a thread to become available and assigns that thread the Runnable that was passed to method execute.

Using an Executor has many advantages over creating threads yourself. Executors can *reuse existing threads* to eliminate the overhead of creating a new thread for each task and can improve performance by *optimizing the number of threads* to ensure that the processor stays busy, without creating so many threads that the application runs out of resources.

We cannot predict the order in which the tasks will start executing, even if we know the order in which they were created and started.







One of the challenges of multithreaded programming is spotting the errors—they may occur so infrequently that a broken program does not produce incorrect results during testing, creating the illusion that the program is correct.

**Atomicity can be achieved using the synchronized keyword.**

**You should never call sleep while holding a lock in a real application.**

Synchronization is necessary only for **mutable data**, or data that may *change* in its lifetime. If the shared data will not change in a multithreaded program, then it’s not possible for a thread to see old or incorrect values as a result of another thread’s manipulating that data. When you share immutable data across threads, declare the corresponding data fields final to indicate that the values of the variables will *not* change after they’re initialized. This prevents accidental modification of the shared data later in a program, which could compromise thread safety. *Labelling object references as final indicates that the reference will not change, but it does not guarantee that the object itself is immutable—this depends entirely on the object’s properties.* However, it’s still good practice to mark references that will not change as final, as doing so forces the object’s constructor to be atomic—the object will be fully constructed with all its fields initialized before the program accesses it.

**Always declare data fields that you do not expect to change as final. Primitive variables that are declared as final can safely be shared across threads. An object reference that’s declared as final ensures that the object it refers to will be fully constructed and initialized before it’s used by the program, and prevents the reference from pointing to another object.**

In a **producer/consumer relationship**, the **producer** portion of an application generates data and stores it in a shared object, and the **consumer** portion of the application reads data from the shared object. The producer/consumer relationship separates the task of identifying work to be done from the tasks involved in actually carrying out the work. In a multithreaded producer/consumer relationship, a **producer thread** generates data and places it in a shared object called a **buffer**. A **consumer thread** reads data from the buffer. This relationship requires *synchronization* to ensure that values are produced and consumed properly. All operations on mutable data that’s shared by multiple threads (e.g., the data in the buffer) must be guarded with a lock to prevent corruption, as discussed. One example of a common producer/consumer relationship is **print spooling (where data from multiple jobs are put in a queue for the printer to print one by one)**.

Operations on the buffer data shared by a producer and consumer thread are also **state dependent**—the operations should proceed only if the buffer is in the correct state. If the buffer is in a *not-full state*, the producer may produce; if the buffer is in a *not-empty state*, the consumer may consume. All operations that access the buffer must use synchronization to ensure that data is written to the buffer or read from the buffer only if the buffer is in the proper state. If the producer attempting to put the next data into the buffer determines that it’s full, the producer thread must *wait* until there’s space to write a new value. If a consumer thread finds the buffer empty or finds that the previous data has already been read, the consumer must also *wait* for new data to become available.