MultiThreading

Chapter 23 Of Java How to Program

Processes and threads

- process: A program running on the computer.
  - Processes have memory isolation (don't share data with each other).
- thread: A "lightweight process"; a single sequential flow of execution or isolated sub-task within one program.
  - A means to implement programs that seem to perform multiple tasks simultaneously (a.k.a. concurrency).

Processes and threads

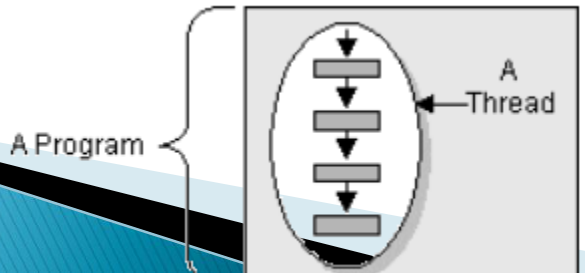
- Threads within the same process do share data with each other.
  - i.e., Variables created in one thread can be seen by others.
  - "shared-memory concurrency“
- sometimes called a lightweight process

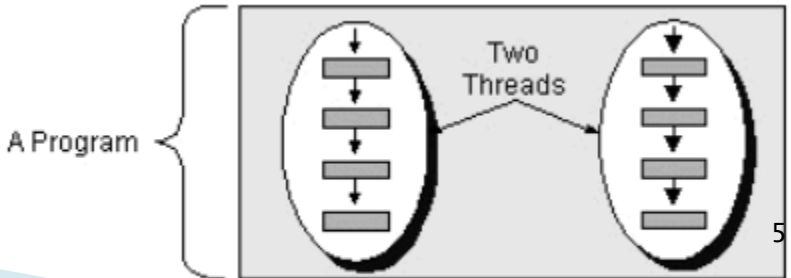
Places threads are used

- I/O:
  - loading a file in the background
- Networking:
  - example: thread that waits for another machine to connect
- Parallelized algorithms
  - example: multithreaded merge sort
- Event-handling loops
  - largely handled for us by Java

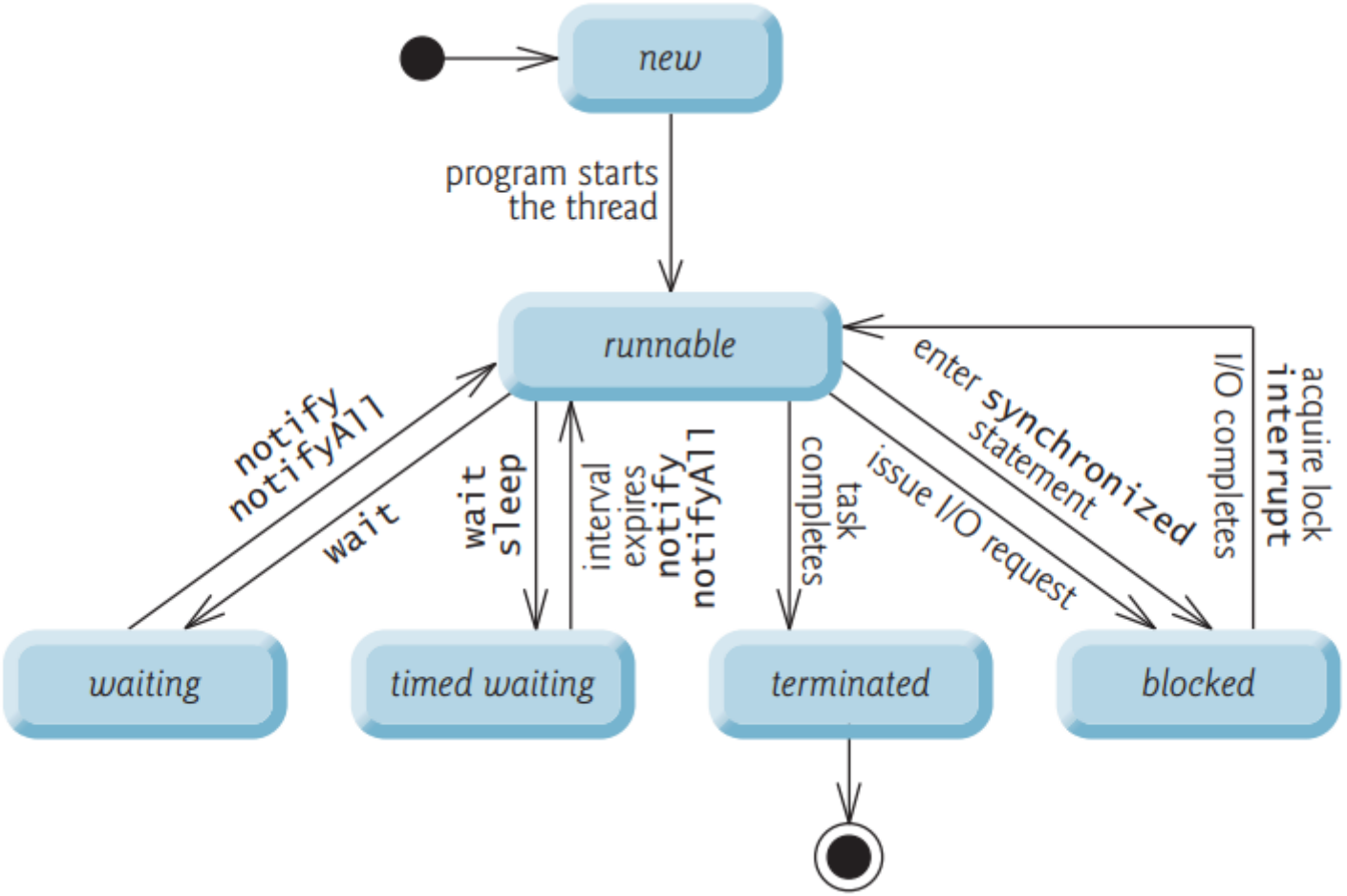
A multithreaded program

- 1 thread:
  - program executes sequentially
  - every program has a "main thread" for its *main* method
- 2 or more threads:
  - runs each thread sequentially, but interleaves them
  - overall program is concurrent
- Scheduling: OS lets each thread/process run for a short time slice then switches to another.
  - High priority threads run first.





Thread life-cycle



Example: Extending from Thread Class

- See Thread Example Code.

Example: Implementing Runnable Interface

- If we need multithreading support in a class that already extends a class other than **Thread**, we must implement the ***Runnable*** *interface* in that class, because Java does not allow a class to extend more than one class at a time.

- See Runnable Example Code.

Example: Thread Pool and ExecutorService

- An ***Executor*** object executes Runnables.
- It does this by creating and managing a group of threads called a ***thread pool***.
- 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.

Example: Thread Pool and ExecutorService

- See Executor Example Code.

```
// create ExecutorService to manage threads
ExecutorService executorService = Executors.newCachedThreadPool();
```

Thread Synchronization

- See Section 23.4
- Threads working on shared objects.
- Multiple threads are updating the shared objects.
- One thread read and one thread update.
- Problem solved by thread synchronization.
- This solution ensures mutual exclusion on working threads.

Thread Synchronization

- A common way to perform synchronization is to use Java's built-in monitors. Every object has a monitor and a monitor lock (or intrinsic lock).

```
synchronized (object)
{
    statements
}
```

```
public synchronized String toString()
{
    return Arrays.toString(array);
}
```

Unsynchronized Data Sharing Example

- See UnsynchronizedExample code.

Synchronized Data Sharing Example

- See SynchronizedExample code.
- Be careful with add and toString methods' signature in SimpleArray class.

Producer/Consumer Relationship

- See ProdConsumExample code.
- See Section 23.6
- 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.

Producer/Consumer Relationship Wait/Notify

- See WaitNotifyExample code.
- See Section 23.7
- We need a way to allow our threads to *wait*, depending on whether certain conditions are true.

Concurrent Collections

Collection	Description
`ArrayBlockingQueue`	A fixed-size queue that supports the producer/consumer relationship—possibly with many producers and consumers.
`ConcurrentHashMap`	**A hash-based map (similar to the `HashMap` introduced in Chapter 16) that allows an arbitrary number of reader threads and a limited number of writer threads. This and the `LinkedBlockingQueue` are by far the most frequently used concurrent collections.**
`ConcurrentLinkedDeque`	A concurrent linked-list implementation of a double-ended queue.
`ConcurrentLinkedQueue`	A concurrent linked-list implementation of a queue that can grow dynamically.
`ConcurrentSkipListMap`	A concurrent map that is sorted by its keys.
`ConcurrentSkipListSet`	A sorted concurrent set.
`CopyOnWriteArrayList`	A thread-safe `ArrayList`. Each operation that modifies the collection first creates a new copy of the contents. Used when the collection is traversed much more frequently than the collection's contents are modified.
`CopyOnWriteArraySet`	A set that's implemented using `CopyOnWriteArrayList`.
`DelayQueue`	A variable-size queue containing `Delayed` objects. An object can be removed only after its delay has expired.

Concurrent Collections

`LinkedBlockingDeque`	A double-ended blocking queue implemented as a linked list that can optionally be fixed in size.
`LinkedBlockingQueue`	**A blocking queue implemented as a linked list that can optionally be fixed in size. This and the `ConcurrentHashMap` are by far the most frequently used concurrent collections.**
`LinkedTransferQueue`	A linked-list implementation of interface `TransferQueue`. Each producer has the option of waiting for a consumer to take an element being inserted (via method `transfer`) or simply placing the element into the queue (via method put). Also provides overloaded method `tryTransfer` to immediately transfer an element to a waiting consumer or to do so within a specified timeout period. If the transfer cannot be completed, the element is not placed in the queue. Typically used in applications that pass messages between threads.
`PriorityBlockingQueue`	A variable-length priority-based blocking queue (like a `PriorityQueue`).
`SynchronousQueue`	[For experts.] A blocking queue implementation that does not have an internal capacity. Each insert operation by one thread must wait for a remove operation from another thread and vice versa.