

# MultiThreading

Chapter 23 Of Java How to Program

Edited by Ehsan Edalat and Amir Kalbasi



# 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 mean 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"
- Processes don't share memory space
  - OS prevents processes to access another process's memory

# Places threads are used

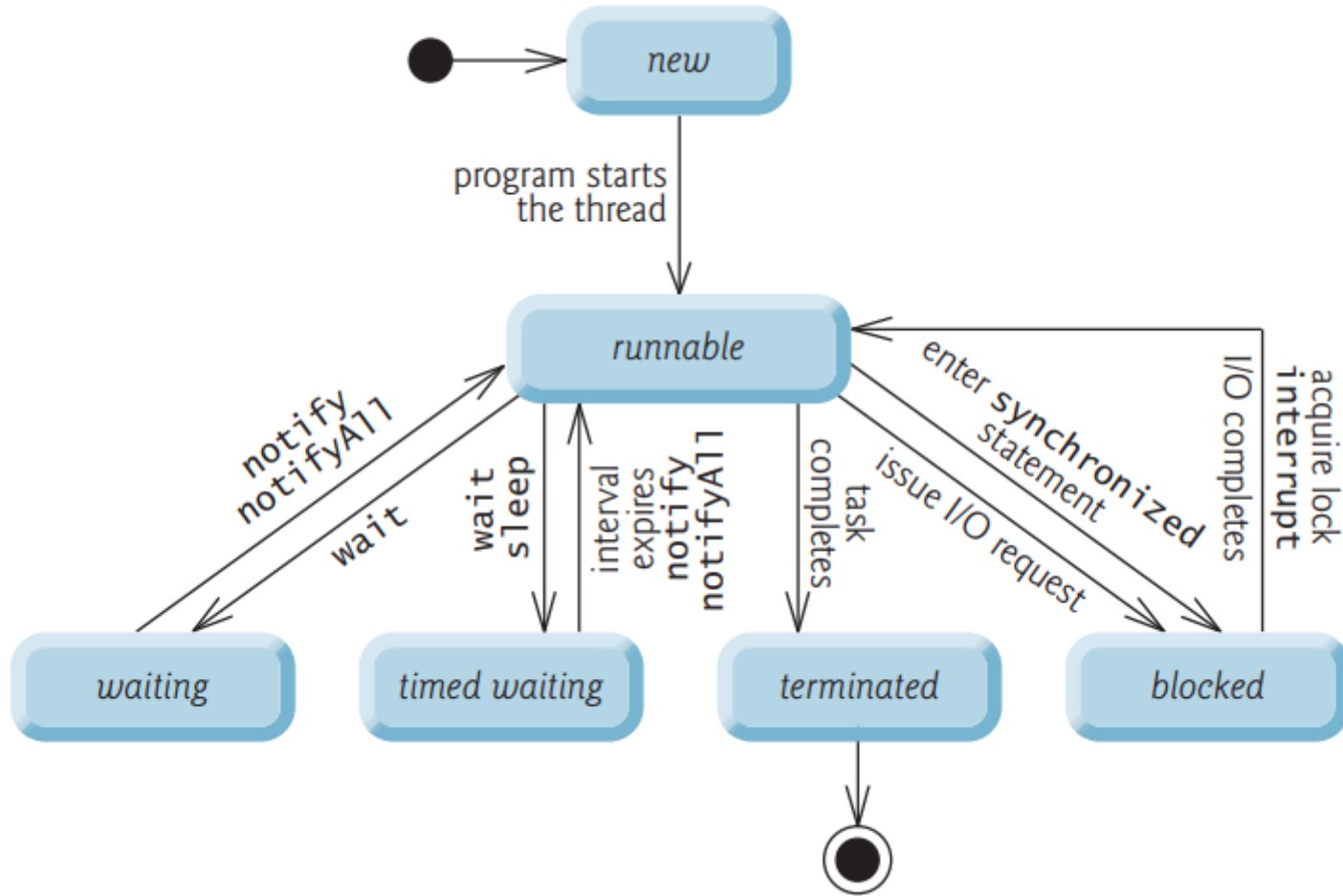
- I/O:
  - loading a file in the background
- Networking:
  - example: thread that waits for another machine to connect
- Parallelized algorithms
  - example: multithreaded merge sort
- Event-handling loops
  - largely handled for us by Java

# 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 **ExecutorService** object executes `Runnable`s.
- 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.

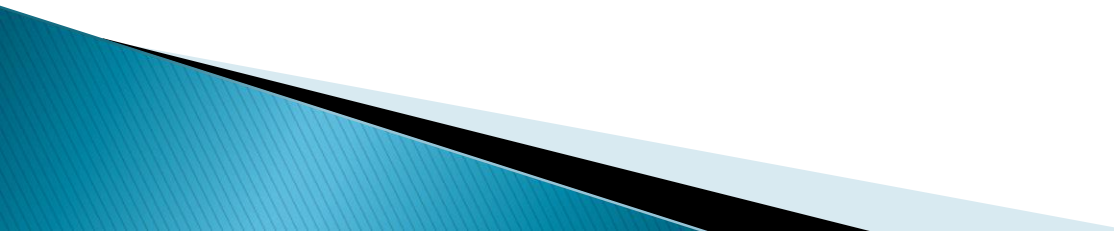
```
// create ExecutorService to manage threads  
ExecutorService executorService = Executors.newCachedThreadPool();
```

# 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.

# Shared Data Example

```
public class Counter {  
    private int c = 0;  
  
    public void increment() {  
        int old = c;  
        c = old + 1;    // c++;  
    }  
  
    public void decrement() {  
        int old = c;  
        c = old - 1;    // c--;  
    }  
  
    public int getValue() {  
        return c;  
    }  
}
```



# Unsynchronized Data Sharing Example

- See `UnsynchronizedExample` code.

# The synchronized keyword

```
// synchronized method: uses "this" object's lock  
public synchronized type name(parameters) {  
    statement(s);  
}
```

```
// synchronized static method: uses the given class's lock  
public static synchronized type name(parameters) {  
    statement(s);  
}
```

```
// synchronized block: uses the given object's lock  
synchronized (object) {  
    statement(s);  
}
```

- ▶ The keyword `synchronized` causes a given method or block of code to acquire an object's lock before running.
  - Ensures that only one thread can be in the given block at a time.

# 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
<code>ArrayBlockingQueue</code>	A fixed-size queue that supports the producer/consumer relationship—possibly with many producers and consumers.
<code>ConcurrentHashMap</code>	A hash-based map (similar to the <code>HashMap</code> introduced in Chapter 16) that allows an arbitrary number of reader threads and a limited number of writer threads. This and the <code>LinkedBlockingQueue</code> are by far the most frequently used concurrent collections.
<code>ConcurrentLinkedDeque</code>	A concurrent linked-list implementation of a double-ended queue.
<code>ConcurrentLinkedQueue</code>	A concurrent linked-list implementation of a queue that can grow dynamically.
<code>ConcurrentSkipListMap</code>	A concurrent map that is sorted by its keys.
<code>ConcurrentSkipListSet</code>	A sorted concurrent set.
<code>CopyOnWriteArrayList</code>	A thread-safe <code>ArrayList</code> . 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.
<code>CopyOnWriteArraySet</code>	A set that's implemented using <code>CopyOnWriteArrayList</code> .
<code>DelayQueue</code>	A variable-size queue containing <code>Delayed</code> objects. An object can be removed only after its delay has expired.

# Concurrent Collections

<code>LinkedBlockingDeque</code>	A double-ended blocking queue implemented as a linked list that can optionally be fixed in size.
<code>LinkedBlockingQueue</code>	A blocking queue implemented as a linked list that can optionally be fixed in size. This and the <code>ConcurrentHashMap</code> are by far the most frequently used concurrent collections.
<code>LinkedTransferQueue</code>	A linked-list implementation of interface <code>TransferQueue</code> . Each producer has the option of waiting for a consumer to take an element being inserted (via method <code>transfer</code> ) or simply placing the element into the queue (via method <code>put</code> ). Also provides overloaded method <code>tryTransfer</code> 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.
<code>PriorityBlockingQueue</code>	A variable-length priority-based blocking queue (like a <code>PriorityQueue</code> ).
<code>SynchronousQueue</code>	[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.

# Multithreading with GUI: SwingWorker

- See SwingWorkerExample code.
- See Section 23.11

Method	Description
<code>doInBackground</code>	Defines a long computation and is called in a worker thread.
<code>done</code>	Executes on the event dispatch thread when <code>doInBackground</code> returns.
<code>execute</code>	Schedules the <code>SwingWorker</code> object to be executed in a worker thread.
<code>get</code>	Waits for the computation to complete, then returns the result of the computation (i.e., the return value of <code>doInBackground</code> ).
<code>publish</code>	Sends intermediate results from the <code>doInBackground</code> method to the <code>process</code> method for processing on the event dispatch thread.

# Multithreading with GUI: SwingWorker

- See `SwingWorkerExample` code.
- See Section 23.11

Method	Description
<code>process</code>	Receives intermediate results from the <code>publish</code> method and processes these results on the event dispatch thread.
<code>setProgress</code>	Sets the progress property to notify any property change listeners on the event dispatch thread of progress bar updates.