Chapter 15 - Threads and Locks

// **Threads in Java**

- every Java thread is created and controlled by a unique object of the java.lang.Thread class
- when a standalone application is run, a user thread is automatically created to execute the main() method → main thread
- we can implement threads in two ways
  - by implementing the java.lang.Runnable interface
  - by extending the java.lang.Thread class

// **Implementing the Runnable interface**

```
public interface Runnable {
	void run();
}
```

- create a class that implements the Runnable interface → Runnable object
- create an object of Type thread by passing a Runnable object as an argument to the constructor
- start() method is invoked on the Thread object created in the previous step

```
public class MyRunnable implements Runnable {
	public int count = 0;

	public void run() {
		System.out.println("RunnableThread starting");
		try {
			while (count < 5) {
				Thread.sleep(500);
				count++;
			}
		} catch (InterruptedException exc) {
			System.out.println("RunnableThread interrupted");
		}
		System.out.println("RunnableThread terminating");
	}
}

public static void main(String[] args) {
        MyRunnable instance = new MyRunnable();
        Thread thread = new Thread(instance);
        thread.start();

        while (instance.count != 5) {
            try {
                Thread.sleep(250);
            } catch (InterruptedException exc) {
                exc.printStackTrace();
            }
        }
    }
```

// **Extending the Thread Class**

```
public class MyThread extends Thread {
    int count = 0;

    public void run() {
        System.out.println("Thread starting");
        try {
            while (count < 5) {
                Thread.sleep(500);
                System.out.println("In Thread, count is " + count);
                count++;
            }
        } catch (InterruptedException exc) {
            System.out.println("Thread interrupted");
        }
        System.out.println("Thread terminating");
    }
}

public class Example {
    public static void main(String[] args) {
        MyThread instance = new MyThread();
        instance.start();

        while (instance.count != 5) {
            try {
                Thread.sleep(250);
            } catch (InterruptedException exc) {
                exc.printStackTrace();
            }
        }
    }
}
```

- Java does not support multiple inheritance → extending the Thread class means that the subclass cannot extending any other class
- inheriting the full overhead of the Thread class might be excessive

// **Synchronization and Locks**

- threads within a given process share the same memory space
- it enables threads to share data
- it also creates the opportunity for issues when two threads modify a resource at the same time
- Java provides synchronization in order to control access to shared resources

// **Synchronized Methods**

- we restrict access to shared resources through the use of the synchronized keyword

```
public class MyClass extends Thread {
    private String name;
    private MyObject myObj;

    public MyClass(MyObject obj, String n) {
        name = n;
        myObj = obj;
    }

    public void run() {
        myObj.foo(name);
    }
}

public class MyObject {
    public synchronized void foo(String name) {
        try {
            System.out.println("Thread " + name + ".foo(): starting");
            Thread.sleep(3000);
            System.out.println("Thread " + name + ".foo(): ending");
        } catch (InterruptedException e) {
            System.out.println("Thread " + name + ": interrupted");
        }
    }
}
```

- can two instances of MyClass call foo at the same time? → if they have the same instance of MyObject then no → if they have different references then yes
- static methods syncronize on the class lock → two threads could not simultaneously execute synchronized static methods on the same class even if they are different

// **Synchronized Blocks**

```
public class MyClass extends Thread {
    ...

    public void run() {
        myObj.foo(name);
    }
}

public class MyObject {
    public void foo(String name) {
        synchronized(this) {
            ...
        }
    }
}
```

- only on thread per instance of MyObject can execute the code within the synchronized block
- if thread1 and thread2 have the same instance of MyObject only one will be allowed to execute the code block at a time

// **Locks**

- locks can be used for more granular control
- a lock (or monitor) is used to synchronize access to a shared resource by associating the resource with the lock
- a thread gets access to a shared resource by first acquiring the lock associated with the resource
- at any given time, at most one thread can hold the lock and therefore only one thread can access the shared resource.

// **Deadlock and Deadlock Prevention**

- a deadlock is a situation where a thread is waiting for an object lock that another thread holds and this second thread is waiting for an object lock that the first thread holds
- in order for a deadlock to occur you must have all of the following conditions met
    - mutual exclusion → limited access to a resource
    - hold and wait → processs already holding a resource can request additional resources, without relinquishing their current resources
    - no preemption → one process cannot forcibly remove another process' resource
    - circular wait → two or more processes form a circular chain where each process is waiting on another resource in the chain

// **Thread vs Process**

- a process is an instance of a program in execution
- a process is an independenty entity to which system resources (CPU, memory) are allocated
- each process is executed in a separate address space
- one process cannot access another process' resources directly → inter-process communication (pipes, files, sockets)
- a thread exists within a process and shares the process' resources (including its heap space)
- each thread has its own registers and its own stack, but other threads can read and write the heap memory

// **Context Switch**

- time spent switching between two processes

// **Notes**

- A lock in Java is owned by the same thread which locked it
- static methods syncronize on the class lock → two threads could not simultaneously execute synchronized static methods on the same class even if they are different

// **FizzBuzz**

- i % 3 == 0 && i % 5 == 0
- i % 3 == 0
- i % 5 == 0