Concurrency

Processes and Threads

- A process is a self-contained program running within its own address space.
- A thread is a single sequential flow of control within a process.
- Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.
- Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic communication.
- Multithreaded execution is an essential feature of the Java platform.
- Every application has at least one thread, called the main thread. This thread has the ability to create additional threads.

Thread Objects

- Each thread is associated with an instance of the class Thread.
- There are two basic strategies for using Thread objects to create a concurrent application.
- The first one provides a Runnable object. The Runnable interface defines a single method, run, meant to contain the code executed in the thread. The Runnable object is passed to the Thread constructor.

Defining and Starting a Thread-Example

```
public class TestThread {
    public static void main(String args[]) {
            Runner1 r = new Runner1();
            Thread t = new Thread(r);
            t.start();

            for(int i=0; i<1000; i++) {
                        System.out.println("Main Thread:------" + i);
            }
    }
}

class Runner1 implements Runnable {
    public void run() {
            for(int i=0; i<1000; i++) {
                        System.out.println("Runner1 :" + i);
            }
    }
}
```

Defining and Starting a Thread

- Subclass Thread. The Thread class itself implements Runnable, though its run method does nothing.
- An application can subclass Thread, providing its own implementation of run.

```
public class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        (new HelloThread()).start();
    }
}
```

Defining and Starting a Thread

- The first idiom, which employs a Runnable object, is more general, because the Runnable object can subclass a class other than Thread.
- The second idiom is easier to use in simple applications, but is limited by the fact that your task class must be a descendant of Thread.
- The Thread class defines a number of methods useful for thread management. These include static methods, which provide information about, or affect the status of, the thread invoking the method.
- The other methods are invoked from other threads involved in managing the thread and Thread object.

Pausing Execution with Sleep

- Thread.sleep causes the current thread to suspend execution for a specified period. This is an efficient means of making processor time available to the other threads of an application or other applications that might be running on a computer system.

```
public class SleepMessages {
    public static void main(String args[])
      throws InterruptedException {
          String importantInfo[] = {  "Mares eat oats", "Does eat oats",
            "Little lambs eat ivy",     "A kid will eat ivy too“        };
          for (int i = 0;  i < importantInfo.length;   i++) {
             //Pause for 4 seconds
             Thread.sleep(4000);
              //Print a message
             System.out.println(importantInfo[i]);
           }
    }
}
```

Interrupts

- An interrupt is an indication to a thread that it should stop what it is doing and do something else.
- It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate.
- A thread sends an interrupt by invoking interrupt on the Thread object for the thread to be interrupted.
- For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption.

Supporting Interruption

```
for (int i = 0; i < importantInfo.length; i++) {
    // Pause for 4 seconds
    try {
        Thread.sleep(4000);
    } catch (InterruptedException e) {
        // We've been interrupted: no more messages.
        return;
    }
    // Print a message
    System.out.println(importantInfo[i]);
}
```

Supporting Interruption

- Many methods that throw InterruptedException, such as sleep, are designed to cancel their current operation and return immediately when an interrupt is received.
- What if a thread goes a long time without invoking a method that throws InterruptedException? Then it must periodically invoke Thread.interrupted, which returns true if an interrupt has been received. For example:

```
for (int i = 0; i < inputs.length; i++) {
    heavyCrunch(inputs[i]);
    if (Thread.interrupted()) {
        // We've been interrupted: no more crunching.
        return;
    }
}
```

Interrupts

- In more complex applications, it might make more sense to throw an InterruptedException:

```
if (Thread.interrupted()) {
    throw new InterruptedException();
}
```

- This allows interrupt handling code to be centralized in a catch clause.

Interrupt-Example

```
import java.util.*;
public class TestInterrupt {
  public static void main(String[] args) {
    MyThread thread = new MyThread();
    thread.start();
    try {
      Thread.sleep(10000);
    }
    catch (InterruptedException e) {}
    thread.interrupt();
  }
}
```

```
class MyThread extends Thread {
  boolean flag = true;
  public void run(){
    while(flag){
      System.out.println("="+new Date()+"=");
      try {
          sleep(1000);
      } catch (InterruptedException e) {
          return;
      }
    }
  }
}
```

Output

```
=Wed Nov 05 21:49:39 CST 2014=
=Wed Nov 05 21:49:40 CST 2014=
=Wed Nov 05 21:49:41 CST 2014=
=Wed Nov 05 21:49:42 CST 2014=
=Wed Nov 05 21:49:43 CST 2014=
=Wed Nov 05 21:49:44 CST 2014=
=Wed Nov 05 21:49:45 CST 2014=
=Wed Nov 05 21:49:46 CST 2014=
=Wed Nov 05 21:49:48 CST 2014=
=Wed Nov 05 21:49:49 CST 2014=
```

The Interrupt Status Flag

- The interrupt mechanism is implemented using an internal flag known as the interrupt status. Invoking Thread.interrupt sets this flag.
- When a thread checks for an interrupt by invoking the static method Thread.interrupted, interrupt status is cleared.
- The non-static isInterrupted method, which is used by one thread to query the interrupt status of another, does not change the interrupt status flag.
- By convention, any method that exits by throwing an InterruptedException clears interrupt status when it does so.
- However, it's always possible that interrupt status will immediately be set again, by another thread invoking interrupt.

Joins

- The join method allows one thread to wait for the completion of another.
- If t is a Thread object whose thread is currently executing,

  ```
  t.join();
  ```

  causes the current thread to pause execution until t's thread terminates.
- Overloads of join allow the programmer to specify a waiting period. However, as with sleep, join is dependent on the OS for timing, so you should not assume that join will wait exactly as long as you specify.
- Like sleep, join responds to an interrupt by exiting with an InterruptedException.

Example

```
public class TestJoin {
  public static void main(String[] args) {
    MyThread2 t1 = new MyThread2("abcde");
    t1.start();
    try {
      t1.join();
    } catch (InterruptedException e) {}

    for(int i=1;i<=5;i++){
      System.out.println("i am main thread");
    }
  }
}
```

```
class MyThread2 extends Thread {
  MyThread2(String s){
       super(s);
  }

  public void run(){
    for(int i =1;i<=5;i++){
      System.out.println("i am "+getName());
      try {
              sleep(1000);
      } catch (InterruptedException e) {
              return;
      }
    }
  }
}
```

Output

```
i am abcde
i am abcde
i am abcde
i am abcde
i am abcde
i am main thread
i am main thread
i am main thread
i am main thread
i am main thread
```

Synchronization

- Threads communicate primarily by sharing access to fields and the objects reference fields refer to.
- This form of communication is extremely efficient, but makes two kinds of errors possible: thread interference and memory consistency errors.
- The tool needed to prevent these errors is synchronization.

Thread Interference

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

Thread Interference

- Counter is designed so that each invocation of increment will add 1 to c, and each invocation of decrement will subtract 1 from c.
- However, if a Counter object is referenced from multiple threads, interference between threads may prevent this from happening as expected.
- Suppose Thread A invokes increment at about the same time Thread B invokes decrement. If the initial value of c is 0, their interleaved actions might follow this sequence:
  1. Thread A: Retrieve c.
  2. Thread B: Retrieve c.
  3. Thread A: Increment retrieved value; result is 1.
  4. Thread B: Decrement retrieved value; result is -1.
  5. Thread A: Store result in c; c is now 1.
  6. Thread B: Store result in c; c is now -1.

Memory Consistency Errors

- Suppose a simple int field is defined and initialized:

```
int counter = 0;
```

- The counter field is shared between two threads, A and B. Suppose thread A increments counter:

```
counter++;
```

- Then, shortly afterwards, thread B prints out counter:

```
System.out.println(counter);
```

- If the two statements had been executed in the same thread, it would be safe to assume that the value printed out would be “1”. But if the two statements are executed in separate threads, the value printed out might well be “0”, because there‘s no guarantee that thread A’s change to counter will be visible to thread B.

Synchronized Methods

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

Synchronized Methods

- If count is an instance of SynchronizedCounter, then making these methods synchronized has two effects:
  - First, it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block suspend execution until the first thread is done with the object.
  - Second, when a synchronized method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

Synchronized Methods

- Synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors: if an object is visible to more than one thread, all reads or writes to that object's variables are done through synchronized methods.
- This strategy is effective, but can present problems with liveness.

Intrinsic Lock

- Synchronization is built around an internal entity known as the intrinsic lock or monitor lock.
- Intrinsic locks play a role in both aspects of synchronization: enforcing exclusive access to an object's state and establishing happens-before relationships that are essential to visibility.
- Every object has an intrinsic lock associated with it.
- By convention, a thread that needs exclusive and consistent access to an object's fields has to acquire the object's intrinsic lock before accessing them, and then release the intrinsic lock when it's done with them.

Intrinsic Lock

- A thread is said to own the intrinsic lock between the time it has acquired the lock and released the lock.
- As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.
- When a thread releases an intrinsic lock, a happens-before relationship is established between that action and any subsequent acquistion of the same lock.
- When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns.
- The lock release occurs even if the return was caused by an uncaught exception.

Intrinsic Lock

- You might wonder what happens when a static synchronized method is invoked, since a static method is associated with a class, not an object. In this case, the thread acquires the intrinsic lock for the Class object associated with the class. Thus access to class's static fields is controlled by a lock that's distinct from the lock for any instance of the class.

Synchronized Statements

- Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

Thread Deadlock

- Deadlock describes a situation where two or more threads are blocked forever, waiting for each other.
- Deadlock occurs when multiple threads need the same locks but obtain them in different order.
- A Java multithreaded program may suffer from the deadlock condition because the **synchronized** keyword causes the executing thread to block while waiting for the lock, or monitor, associated with the specified object.

Example

```
public class TestThread {
   public static Object Lock1 = new Object();
   public static Object Lock2 = new Object();
   public static void main(String args[]) {
      ThreadDemo1 T1 = new ThreadDemo1();
      ThreadDemo2 T2 = new ThreadDemo2();
      T1.start();
      T2.start();
   }
```

Example

```
private static class ThreadDemo1 extends Thread {
    public void run() {
      synchronized (Lock1) {
          System.out.println("Thread 1: Holding lock 1...");
          try { Thread.sleep(10); }
          catch (InterruptedException e) {}
          System.out.println("Thread 1: Waiting for lock 2...");
          synchronized (Lock2) {
            System.out.println("Thread 1: Holding lock 1 & 2...");
          }
      }
    }
}
```

Example

```
private static class ThreadDemo2 extends Thread {
    public void run() {
      synchronized (Lock2) {
        System.out.println("Thread 2: Holding lock 2...");
        try { Thread.sleep(10); }
        catch (InterruptedException e) {}
        System.out.println("Thread 2: Waiting for lock 1...");
        synchronized (Lock1) {
          System.out.println("Thread 2: Holding lock 1 & 2...");
        }
      }
    }
  }
}
```

Output

```
Thread 1: Holding lock 1...
Thread 2: Holding lock 2...
Thread 1: Waiting for lock 2...
Thread 2: Waiting for lock 1...
```

Solutions

```
private static class ThreadDemo2 extends Thread {
    public void run() {
      synchronized (Lock1) {
        System.out.println("Thread 2: Holding lock 1...");
        try { Thread.sleep(10); }
        catch (InterruptedException e) {}
        System.out.println("Thread 2: Waiting for lock 2...");
        synchronized (Lock2) {
          System.out.println("Thread 2: Holding lock 1 & 2...");
        }
      }
    }
  }
```

Output

```
Thread 1: Holding lock 1...
Thread 1: Waiting for lock 2...
Thread 1: Holding lock 1 & 2...
Thread 2: Holding lock 1...
Thread 2: Waiting for lock 2...
Thread 2: Holding lock 1 & 2...
```

Interthread Communication

- Inter thread communication is important when you develop an application where two or more threads exchange some information.
- There are simply three methods and a little trick which makes thread communication possible. First let's see all the three methods listed below:
  - public void wait(): Causes the current thread to wait until another thread invokes the notify().
  - public void notify(): Wakes up a single thread that is waiting on this object's monitor.
  - public void notifyAll():Wakes up all the threads that called wait( ) on the same object.
- These methods have been implemented as final methods in Object, so they are available in all the classes. All three methods can be called only from within a synchronized context.

Example

```
class Chat {
    boolean flag = false;
    public synchronized void Question(String msg) {
        if (flag) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println(msg);
        flag = true;
        notify();
    }
```

Example

```
    public synchronized void Answer(String msg) {
        if (!flag) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println(msg);
        flag = false;
        notify();
    }
}
```

Example

```
class T1 implements Runnable {
    Chat m;
    String[] s1 = { "Hi", "How are you ?", "I am also doing fine!" };
    public T1(Chat m1) {
        this.m = m1;
        new Thread(this, "Question").start();
    }
    public void run() {
        for (int i = 0; i < s1.length; i++) {
            m.Question(s1[i]);
        }
    }
}
```

Example

```
class T2 implements Runnable {
    Chat m;
    String[] s2 = { "Hi", "I am good, what about you?", "Great!" };
    public T2(Chat m2) {
        this.m = m2;
        new Thread(this, "Answer").start();
    }
    public void run() {
        for (int i = 0; i < s2.length; i++) {
            m.Answer(s2[i]);
        }
    }
}
```

Exampe

```
public class TestThread {
    public static void main(String[] args) {
        Chat m = new Chat();
        new T1(m);
        new T2(m);
    }
}
```

When above program is complied and executed, it outputs:

```
Hi
Hi
How are you ?
I am good, what about you?
I am also doing fine!
Great!
```