ID1217 Programming with Processes
Lecture 12

ROYAL INSTITUTE OF TECHNOLOGY

Java Monitors. Concurrent Utilities in Java SDK

Vladimir Vlassov
KTH/ICT/EECS

Outline

- Revisit:
  - Java threads (see also Lecture 7)
  - Monitors (see also Lecture 11)
- Thread synchronization in Java
  - **`synchronized`** methods and blocks
  - Shared objects as monitors; Bounded buffer (consumer-producer) example
- Java concurrent utilities
  - Locks and Conditions;
  - The Executor framework; Example of using a thread pool
  - Synchronizers; Atomic variables; Concurrent collections
- See also Lecture 6 / Threads, locks and Condition Variables in Java
- Further reading:
  - The Java Tutorials. Lesson: Concurrency
    http://docs.oracle.com/javase/tutorial/essential/concurrency/
  - Java Concurrency Utilities
    http://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/index.html

ROYAL INSTITUTE OF TECHNOLOGY

Multithreading in Java

- A Java thread is a light-weight process represented by an object of the `Thread` (sub)class
  - Stack, execution context
  - Access all variables in its scope
- Provides methods
  - `synchronized native void start()`
    - Start this thread
  - `void join()`
    - Wait for this thread to die
  - `void run()`
    - Thread executes when it starts
    - Thread vanishes when it returns
    - You must implement this method

Thread Class and Runnable Interface

```
public class Thread extends Object implements Runnable {
     public Thread();
     public Thread(Runnable target);
     public Thread(String name);
     public Thread(Runnable target, String name);
       ...
     public synchronized native void start();
     public void run();
       ...
   }
public interface Runnable{
   public void run();
}
```

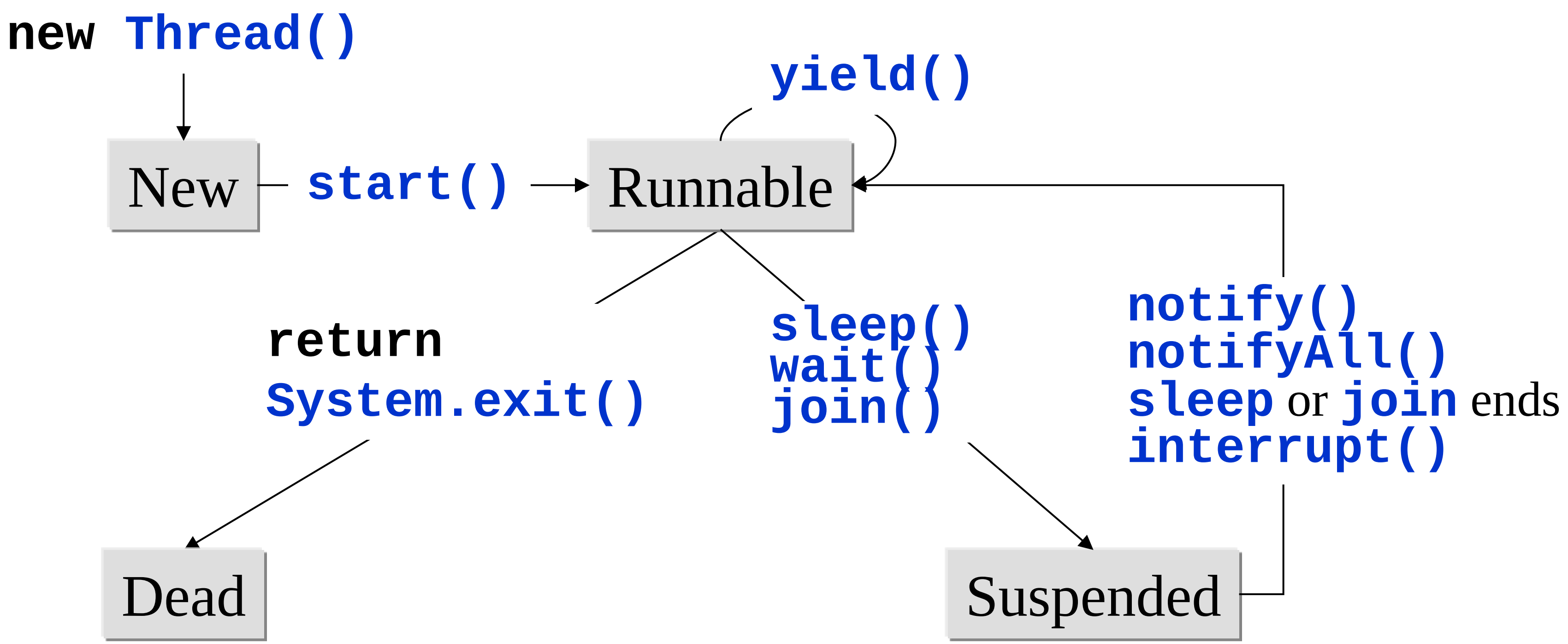
First Way to Program and Create a Java Thread

- Extend the **`Thread`** class
  - Override the **`run`** method and define other methods if needed;
  - Create and start a thread:
    - Instantiate the **`Thread`** subclass;
    - Call the **`start`** method on the thread object – creates a thread context and invokes **`run`** to be executed in a separate thread
- See examples in Lecture 6.

Another Way to Program and Create Java Threads

- Implement the **`Runnable`** interface in a class that represent a class of ***tasks*** to be execute in a thread
  - Develop a class with the **`Runnable`** interface
    - It provides a **`run()`** method that does what you want;
  - Create an object of that class with the **`Runnable`** interface;
  - Create a thread passing the **`Runnable`** object to a thread constructor;
  - Start the thread.
- See examples in Lecture 6.

Thread State Diagram



- IO operations affect states Runnable and Suspended in the ordinary way

Thread Interactions

- Threads in Java execute concurrently – at least conceptually.
- Threads can interact and communicate
  - **Via shared objects**.
  - By calling methods and accessing variables of each other like ordinary objects;
  - Via pipes, sockets.
- An object is **shared** when concurrent threads have references to it and can invoke its methods or access its variables.

Thread Interactions (cont'd)

- Java provides a number of ways to synchronize access to shared objects, both built-in (`synchronized`) and through packages (`java.util.concurrent`).
- First we describe the built-in model, called the ***monitor*** model, which is the simplest and most commonly-used approach.
- Next we consider Java concurrent utilities

Remind: Monitors

- A **monitor** is a programming language construct (e.g. a class) which encapsulates variables, access procedures and initialization code within an abstract data type.
- **Monitor variables** hold a state of an instance of the monitor;
- **Monitor procedures** (a.k.a. operations or methods) are the only means to access (to inspect and to alter) the monitor variables;
- **Monitor procedures are executed with mutual exclusion**.
  - Every instance of a monitor has a unique lock, a.k.a the entry lock or the monitor lock
- Monitors were considered in Lecture 11

The Monitor Model in Java

- A shared object may have ***synchronized methods or code blocks*** to be executed ***with mutual exclusion***
- The **`synchronized`** modifier defines mutual exclusion for an entire method or a code block
- ***Java monitors*** are objects with synchronized methods

`synchronized` Methods and Blocks

- `synchronized` methods and blocks are executed with mutual exclusion, i.e. by one thread at a time
- Each object with **synchronized methods** has its own **implicit lock** associated with the object, i.e. the object itself serves as a lock

```
public synchronized void put (int number) {
   //code to be executed with mutual exclusion
}
```

- Each **synchronized block** requires **a lock object to be explicitly indicated**
  - Any object can be used as a lock for a synchronized block

```
Object lock = new Object();
...
synchronized (lock) { // the object "lock" is used as a lock
   //code to be executed with mutual exclusion
}
```

`synchronized` Methods and Blocks (cont'd)

- A thread obtains the lock when calling a synchronized method or executing a synchronized block
- A thread may hold locks of more then one objects, i.e. nested synchronized calls are closed

Wait Set in `synchronized` Methods and Blocks

- Beside the lock, each synchronized object or block has also **one implicit condition variable** called **wait set** associated with the lock
- A thread may wait on and signal to the wait set by calling `wait()`, `notify()` and `notifyAll()` in scope of a `synchronized` method or block

synchronized Method

```
public class ComputeMax {

  private int max = Number.MIN_VALUE;

  public synchronized int getMax(int value) {
    if (value > max) max = value;
    return max;
  }
}
```

synchronized method

synchronized Block

```
public class ComputeMax {
   private int max = Number.MIN_VALUE;

   public int getMax(int value) {
      if (value > max) {
         synchronized (this) {
            if (value > max) max = value;
         }
      }
      return max;
   }
}
```

synchronized block

Monitors in Java

- **Java monitor** is an object of a class with **`synchronized`** methods, which can be invoked by one thread at a time.
  - A class may contain synchronized and ordinary non-synchronized methods – the latter are executed without synchronization.
- Each monitor has an implicit **monitor lock**
- Each monitor has an implicit **condition variable** (a.k.a. **wait set**)
  - **`wait()`**, **`notify()`** and **`notifyAll()`** in scope of a **`synchronized`** method;
  - No priority wait;
  - Signal-and-Continue policy of **`notify()`** and **`notifyAll()`**

Java Synchronized Methods (1/5)

```
public class Queue<T> {

  int head = 0, tail = 0;
  T[QSIZE] items;

  public synchronized T deq() {
   while (tail – head == 0)
     this.wait();
   T result = items[head % QSIZE]; head++;
   this.notifyAll();
   return result;
  }
  …
}}
```

Java Synchronized Methods (2/5)

```
public class Queue<T> {

  int head = 0, tail = 0;
  T[QSIZE] items;

  public synchronized T deq() {
   while (tail – head == 0)
     this.wait();
   T result = items[head % QSIZE]; head++;
   this.notifyAll();
   return result;
  }
  …
}}
```

Each object has an implicit lock with an implicit condition

Java Synchronized Methods (3/5)

```
public class Queue<T> {

  int head = 0, tail = 0;
  T[QSIZE] items;

  public synchronized T deq() {
   while (tail – head == 0)
     this.wait();
   T result = items[head % QSIZE]; head++;
   this.notifyAll();
   return result;
  }
  …
}}
```

Lock on entry, unlock on return

Java Synchronized Methods (4/5)

```
public class Queue<T> {

  int head = 0, tail = 0;
  T[QSIZE] items;

  public synchronized T deq() {
   while (tail – head == 0)
     this.wait();
   T result = items[head % QSIZE]; head++;
   this.notifyAll();
   return result;
  }
  …
}}
```

Wait on implicit condition

Java Synchronized Methods (5/5)

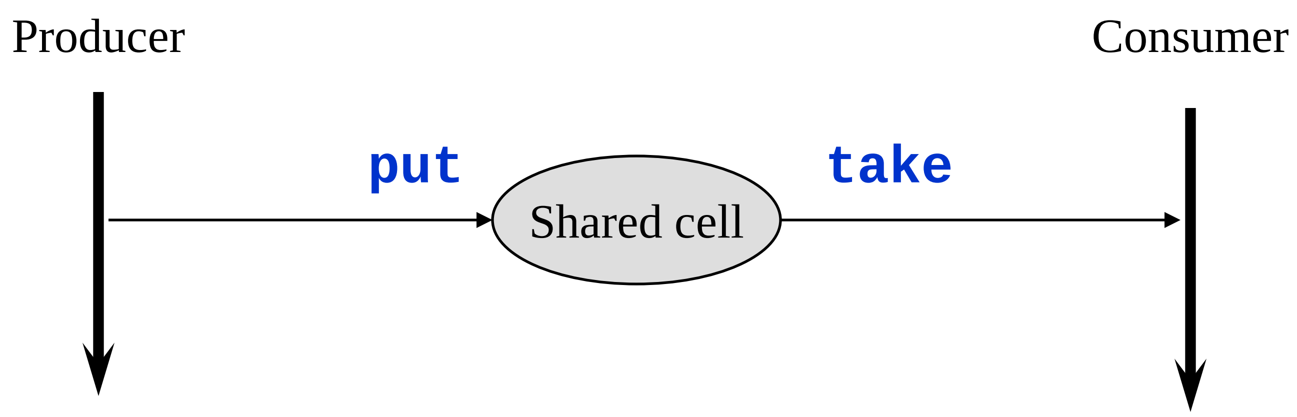
ROYAL INSTITUTE OF TECHNOLOGY

```
public class Queue<T> {

  int head = 0, tail = 0;
  T[QSIZE] items;

  public synchronized T deq() {
   while (tail – head == 0)
     this.wait();
   T result = items[head % QSIZE]; head++;
   this.notifyAll();
   return result;
  }
  …
}}
```

Signal all threads waiting on condition

A Simple Example: Producer/Consumer

- Producer and Consumer threads are using a shared object (Shared Cell monitor) to interact in a dataflow fashion



- The "shared cell" (buffer) is a monitor
  - Methods **`put`** and **`take`** are synchronized to be executed with mutual exclusion.
  - An implicit condition variable ("wait set") is used for condition synchronization of Producer and Consumer.

The Shared Cell Monitor

```
public class SharedCell {
  private int value;
  private boolean empty = true;
  public synchronized int take() {
    while (empty) {
      try {
        wait ();
      } catch (InterruptedException e) { }
    }
    empty = true;
    notify ();
    return value;
  }
  public synchronized void put(int value) {
    while (!empty) {
      try {
        wait ();
      } catch (InterruptedException e) { }
    }
    this.value = value;
    empty = false;
    notify ();
  }
}
```

- A test application – the primary class **Exchange**:

```
public class Exchange {
  public static void main(String args[])
    {
    SharedCell cell = new SharedCell ();
    Producer p = new Producer (cell);
    Consumer c = new Consumer (cell, 10);
    p.start ();
    c.start ();
    try {
      c.join ();
    } catch (InterruptedException e) { };
    p.setStop ();
    p.interrupt();
  }
}
```

Producer and Consumer Classes

```
class Producer extends Thread {
  private SharedCell cell;
  private boolean Stop = false;
  public Producer (SharedCell cell) {
    this.cell = cell;
  }
  public void setStop () {
    Stop = true;
  }
  public void run () {
    int value;
    while (!Stop) {
    value = (int) (Math.random () * 100);
    cell.put (value);
    try {
      sleep (value);
    } catch (InterruptedException e) { }
     }
  }
}
```

```
class Consumer extends Thread {
  private SharedCell cell;
  private int n;
  public Consumer(SharedCell cell, int n) {
    this.cell = cell;
    this.n = n;
  }
  public void run () {
    int value;
    for (int i = 0; i < n; i++) {
      value = cell.take ();
      System.out.println ("Consumer: " + i +
     " value = " + value);
    }
  }
}
```

Example: Synchronized Bounded Buffer

```
public class Bounded_Buffer {
   private Object[] items;
   private int count = 0, front = 0, rear = 0;
   private int n;

   public Bounded_Buffer(int n) {
       this.n = n;
       items = new Object[n];
   }
```

Synchronized Bounded Buffer (cont'd)

```
   public synchronized void put(Object x) {
      while (count == n)
         try { wait(); }
         catch (InterruptedException e) { }
      items[rear] = x; rear = (rear + 1) % n; count++;
      notifyAll();
   }
   public synchronized Object take() {
      while (count == 0)
         try { wait(); }
         catch (InterruptedException e) { }
      Object x = items[front];
      front = (front + 1) % n; count--;
      notifyAll();
      return x;
   }
}
```

Java Concurrency Utilities: `java.util.concurrent`

- **Locks** and **Conditions**
- **Synchronizers**
  - General purpose synchronization classes, including semaphores, mutexes, barriers, latches, and exchangers
- **The Executor framework**
  - for scheduling, execution, and control of asynchronous tasks (`Runnable` objects)
- **Nanosecond-granularity timing**
  - The actual precision of `System.nanoTime` is platform-dependent
  - Used for time-stamps and time estimates

Concurrency Utilities: (cont'd)

ROYAL INSTITUTE OF TECHNOLOGY

- **Atomic Variables**
  - Classes for atomically manipulating single variables (of primitive types and references)
  - E.g. **AtomicBoolean**, **AtomicInteger**, **AtomicLong**
  - For object references and arrays
  - E.g. **AtomicReference<V>**, **AtomicMarkableReference<V>**, **AtomicStampedReference<V>**
  - Used to implement concurrent collection classes
- **Concurrent Collections**
  - Pools of items
  - **Queue** and **BlockingQueue** interfaces
  - Concurrent implementations of **Map**, **List**, and **Queue**.

ROYAL INSTITUTE OF TECHNOLOGY

Locks and Conditions

- `java.util.concurrent.locks`
  - Classes and interfaces for locking and waiting for conditions
- `ReentrantLock` class
  - Represents a reentrant mutual exclusion lock
  - Allows to create condition variables to wait for conditions
- `Condition` interface
  - Represents a condition variable associated with a lock
  - Allows one thread to suspend execution (to "wait") until notified by another thread
  - The suspended thread releases the lock
- `ReentrantLock` locks (like `synchronized` objects) are monitors
  - Allow blocking on a condition rather than spinning
- Threads:
  - acquire and release lock
  - wait on a condition

The Java Lock Interface

```
public interface Lock {
 void lock();
 void lockInterruptibly() throws InterruptedException;
 boolean tryLock();
 boolean tryLock(long time, TimeUnit unit);
 Condition newCondition();
 void unlock;
}
```

Lock Conditions

```
public interface Condition {
  void await();
  boolean await(long time, TimeUnit unit);
  …
  void signal();
  void signalAll();
 }
```

Await, Signal and Signal All

```
q.await()
```

- Releases lock associated with `q`
- Sleeps (gives up processor)
- Awakens (resumes running) when signaled by `Signal` or `SignalAll`
- Reacquires lock & returns

```
q.signal();
```

- Awakens **one** waiting thread
  - Which will reacquire lock associated with `q`

```
q.signalAll();
```

- Awakens **all** waiting threads
  - Which will each reacquire lock associated with `q`

Example: Lock-Based Blocking Bounded Buffer

```
public class BoundedBuffer {
  final Lock lock = new ReentrantLock();
  final Condition notFull  = lock.newCondition();
  final Condition notEmpty = lock.newCondition();
  final Object[] items;
  int rear, front, count, n;

  public BoundedBuffer(int n) {
     this.n = n;
     buf = new Object[n];
  }
```

```
  public void put(Object x) throws InterruptedException {
    lock.lock();
    try {
      while (count == n) notFull.await();
      items[rear] = x; rear = (rear + 1) % n; count++;
      notEmpty.signal();
    } finally {
      lock.unlock();
    }
  }
  public Object take() throws InterruptedException {
    lock.lock();
    try {
      while (count == 0) notEmpty.await();
      Object x = items[front];
      front = (front + 1)% n; count--;
      notFull.signal();
      return x;
    } finally {
      lock.unlock();
    }
  }
}
```

Synchronizers

- **`java.util.concurrent`**
  - Utility classes commonly useful for synchronization
- **`Semaphore`**
  - **`acquire()`** – P operation
  - **`release()`** – V operation
- **`CyclicBarrier`**
  - A reusable barrier
  - Allows a set of threads to wait for each other at a barrier point
  - **`await()`**
  - Useful for parallel programming (iterative parallelism)

Synchronizers (cont'd)

- **`CountDownLatch`**
  - Allows one or more threads to wait until a set of operations (signals, events, or conditions) being performed in other threads completes
  - **`countDown()`**, **`await()`**, **`getCount()`**
  - Can be used instead of barrier or join
  - Useful for master-worker applications
    - See examples in the Java SE API documentation
- **`Exchanger<V>`**
  - Allows two threads to exchange objects at a rendezvous point
  - **`public V exchange(V x)`**
  - Useful for pipeline parallelism
    - See examples in the Java SE API documentation

Example: Master-Workers Using Latches

- Master and Worker threads synchronize using two counting down latches
  - **`startSignal`** – Master lets Workers to execute
  - **`doneSignal`** – A Worker informs Master that it finished
- Master creates and starts worker threads, which are waiting for a ”start” signal
- Master lets worker threads to run via the **`startSignal`** latch, and waits for all to finish
- Each worker, when it’s done, signals the Master via the **`doneSignal`** latch

Example: Master-Workers Using Latches

```java
class Master { // ...
  public static void main() throws InterruptedException {
    CountDownLatch startSignal = new CountDownLatch(1);
    CountDownLatch doneSignal = new CountDownLatch(N);
    for (int i = 0; i < N; ++i) // create and start threads
      new Thread(new Worker(startSignal, doneSignal)).start();
    doSomethingElse(); // don't let run yet
    startSignal.countDown(); // let all threads proceed
    doSomethingElse();
    doneSignal.await(); // wait for all to finish
  }
}
class Worker implements Runnable {
  private final CountDownLatch startSignal, doneSignal;
  Worker(CountDownLatch startSignal, CountDownLatch doneSignal) {
    this.startSignal = startSignal;
    this.doneSignal = doneSignal;
  }
  public void run() {
    try {
      startSignal.await(); // wait for a start signal from master
      doWork(); // perform my task
      doneSignal.countDown(); // signal master that I am done
    } catch (InterruptedException ex) {}
    return;
  }
}
```

The Executor Framework

- For scheduling, execution, and control of asynchronous tasks in concurrent threads according to a set of execution policies
- Allows creating an executor (a pool of threads) and assigning tasks to the executor to execute
- An **`Executor`** object executes submitted tasks
- For example:

```
Executor e =
        Executors.newFixedThreadPool(numThreads);
e.execute(new RunnableTask1());
e.execute(new RunnableTask2());
```

Executor Interfaces

- An executor can have one of the following interfaces:
- **`Executor`**
  - A simple interface to launch void Runnable tasks
  - **`execute(Runnable)`**
- **`ExecutorService`**
  - Executor subinterface with additional features to manage lifecycle
  - To launch and control void Runnable tasks and Callable tasks, which return results
  - **`submit(Runnable)`** , **`submit(Callable<T>)`**, **`shutdown()`**, **`invokeAll(...)`**, **`awaitTermination(…)`**
  - **`Future<V>`** represents the result of an asynchronous computation
- **`ScheduledExecutorService`**
  - ExecutorService subinterface with support for future or periodic execution
  - For scheduling Runnable and Callable tasks

Example: Using an Executer (a Thread Pool)

```
import java.io.*;
import java.net.*;
public class Handler implements Runnable {
   private Socket socket;
   public Handler(Socket socket) {  this.socket = socket; }
   public void run() {
      try {
         BufferedReader rd = new BufferedReader(
                   new InputStreamReader(socket.getInputStream()));
         PrintWriter wr = new PrintWriter(socket.getOutputStream());
         String str;
         while ((str = rd.readLine()) != null) {
            for ( int i=str.length(); i > 0; i-- ) wr.print(str.charAt(i-1));
               wr.println();
               wr.flush();
            }
            socket.close();
         } catch ( IOException e ) {;}
    }
}
```

```java
import java.io.*;
import java.net.*;
import java.util.concurrent.*;
public class ReverseServer {
    public static void main(String[] args) throws IOException {
        int poolSize = 3, port = 4444;
        ServerSocket serverSocket = null;
        try {
           if (args.length >1) poolSize = Integer.parseInt(args[1]);
           if (args.length >0) port = Integer.parseInt(args[0]);
        } catch (NumberFormatException e) {
           System.out.println("USAGE: java ReverseServer [poolSize] [port]");
           System.exit(1);
        }
        try {
            serverSocket = new ServerSocket(port);
        } catch (IOException e) {
            System.out.println("Can not listen on port: " + port);
            System.exit(1);
        }
        ExecutorService executor = Executors.newFixedThreadPool(poolSize);
        while (true) {
            Socket socket = serverSocket.accept();
            executor.execute( new Handler(socket) );
        }
    }
}
```

ROYAL INSTITUTE OF TECHNOLOGY

Atomic Variables

- `java.util.concurrent.atomic`
  - Classes that support atomic operations on single variables.
- For primitive type vars and arrays
  - E.g. `AtomicBoolean`, `AtomicInteger`, `AtomicLong`
- For object references and arrays
  - E.g. `AtomicReference<V>`, `AtomicMarkableReference<V>`, `AtomicStampedReference<V>`
- **Used to implement concurrent collections**

Each Atomic Class Has

- Atomic **`get`** and **`set`** methods
  - Get returns the last set value
- Atomic **`compareAndSet`** method (a.k.a. **CAS**)
  - An atomic conditional update of the form

```
boolean compareAndSet(expectedValue, updateValue);
```

  - Atomically set the value to the given updated value if the current value is equal to the expected value;
  - Used to implement concurrent collections, concurrent objects;
  - The CAS operation has infinite consensus number (i.e. it allows to solve the wait-free consensus problem for an arbitrary number of threads)

Examples of Using Atomic Variables

- A sequence number generator

```
class Sequencer {
  private AtomicLong seqNumber = new AtomicLong(0);
  public long next() {
     return seqNumber.getAndIncrement();
  }
}
```

- Atomic counter

```
class AtomicCounter {
  private AtomicInteger cnt = new AtomicInteger(0);
  public void increment() { cnt.incrementAndGet(); }
  public void decrement() { cnt.decrementAndGet(); }
  public int value() { return cnt.get(); }
}
```

Java Collections Framework

- The **Java collections framework** (package **`java.util`**)
  - Includes collection interfaces and classes, e.g. **`HashSet<E>`**, **`LinkedList<E>`**

- A **collection** is an object that represents a group of elements (objects) of a specified type, i.e. **`Vector<E>`**
  - Operations (depend on collection type): add, remove, put, replace, get, peek, poll, contains, size, list, isEmpty, etc.

- **Concurrent Collections** (**`java.util.concurrent`**)
  - Extends the Java Collection framework (**`java.util`**) with concurrent collections including the **`Queue`**, **`BlockingQueue`** and **`BlockingDeque`** interfaces, and high-performance, concurrent implementations of **`Map`**, **`List`**, and **`Queue`**.

General Purpose Collections in `java.util`

Interfaces	Collection classes <*Implementation-style*><*Interface*>				
	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
`Set<E>`	**`HashSet`**		**`TreeSet`**		**`LinkedHashSet`**
`List<E>`		**`ArrayList`**		**`LinkedList`**	
`Deque<E>`		**`ArrayDeque`**		**`LinkedList`**	
`Map<K,V>`	**`HashMap`**		**`TreeMap`**		**`LinkedHashMap`**

Synchronized Collections

- Collections in **`java.util`** are not synchronized
- To make a **synchronized (thread-safe) collection**:

```
Collection<T> Collections.synchronizedCollection(Collection<T> c)
```

  – Returns a synchronized collection backed by the specified collection.
  – The synchronized collection should be used as **a lock for all accesses** to the backing collection
  – For example:

```
Collection c = Collections.synchronizedCollection(myCollection);
...
synchronized(c) {
  Iterator i = c.iterator(); // Must be in the synchronized block
  while (i.hasNext())
     foo(i.next());
}
```

Concurrent Collections (`java.util.concurrent`)

- Concurrent versions of some collections
  - `ConcurrentHashMap<K,V>`
  - `CopyOnWriteArrayList`
  - `CopyOnWriteArraySet`
- Different from similar "synchronized" classes
- **A concurrent collection is thread-safe, but not governed by a single exclusion lock.**
  - For example, `ConcurrentHashMap`, safely permits any number of concurrent reads as well as a tunable number of concurrent writes.

Unsynchronized, Synchronized, Concurrent Collections

- When to use which
- Unsynchronized collections
  - preferable when either collections are unshared, or are accessible only when holding other locks.
- ”Synchronized” versions
  - when you need to govern all access to a collection via a single lock, at the expense of poorer scalability.
- ”Concurrent" versions
  - normally preferable when multiple threads are expected to access a common collection.