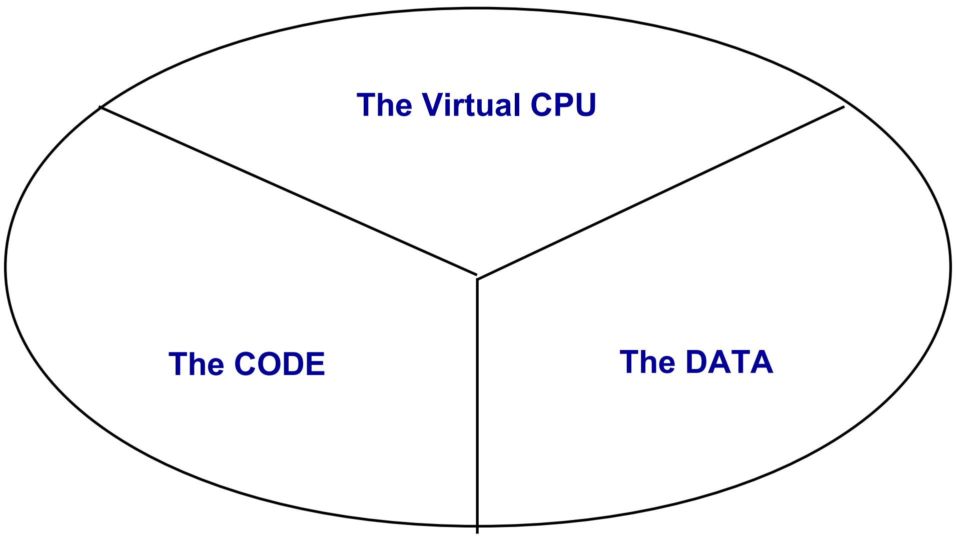
Threads

What is a thread ?

- The encapsulation of a virtual CPU with its own code and data.

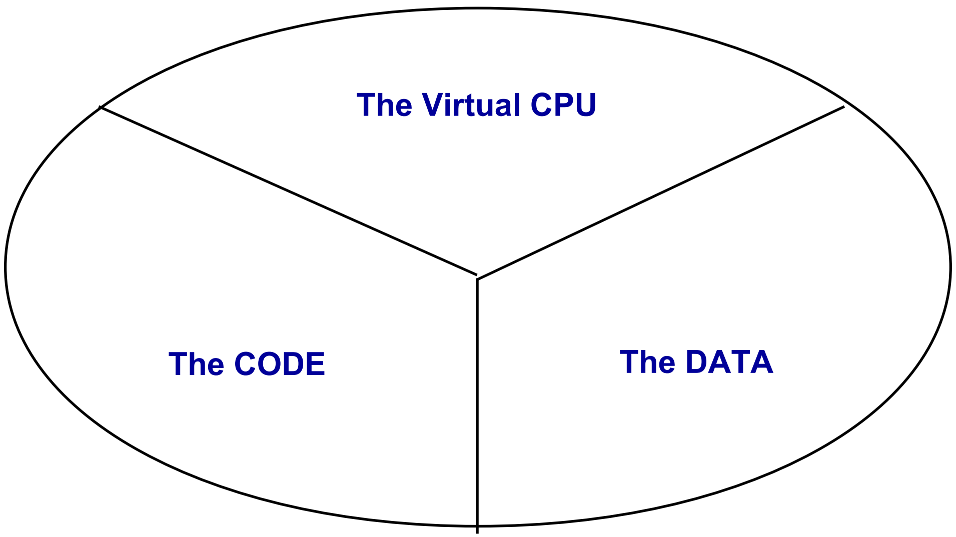
The Threads Parts

- Each thread is composed of the following three parts:



The Threads Parts

- Each thread is composed of the following three parts:



Creating New Thread

- When creating a new thread, besides the Thread object, there is a need in a Runnable object.

- The class Thread has a constructor that receives a Runnable reference.

```
Runnable ob = new Greengrocer("BANANA");
Thread t = new Thread(ob);
```

- Now it is clear that the virtual CPU is the Thread object, the CODE is one that starts in the run() method which was declared in Greengrocer class and the DATA is ob.

Starting The New Thread

- Only when calling the start() method on the Thread object, the new thread starts working.
  t.start();

- The thread starts running concurrently with the thread during which the start() method was called.

Simple Example

- The following is a simple example that presents three concurrent threads:

```
public class SimpleThreadDemo
{
        public static void main(String args[])
        {
                Thread t1, t2;
                Greengrocer g1, g2;
                g1 = new Greengrocer("BANANA");
                g2 = new Greengrocer("TOMATO");
                t1 = new Thread(g1);
                t2 = new Thread(g2);
```

Simple Example

```
            t1.start();
            t2.start();
            for(int i=0; i<20; i++)
            {
                  System.out.println("MARKET, current
                  thread is : " +
                  Thread.currentThread().getName());
                  try  {Thread.sleep(100);}
                  catch(InterruptedException e){}
            }
      }
}
```

A Simple Example

```
class Greengrocer implements Runnable
{
    private String product;

    Greengrocer(String str)
    {
            product = str;
    }
```

A Simple Example

```
    public void run()
    {
            for(int i=0; i<20; i++)
            {
                    System.out.println(product
    + ",current thread is : "
    + Thread.currentThread().getName());
                    try
                    {
                            Thread.sleep(100);
                    }
                    catch(InterruptedException e) {}
            }
    }
}
```

Another Simple Example

- The following is an example for two threads that share the same data:that presents three concurrent

```
public class AnotherSimpleThreadDemo
{
        public static void main(String args[])
        {
                Thread t1, t2;
                Greengrocer g1, g2;
                g1 = new Greengrocer("BANANA");
                g2 = new Greengrocer("TOMATO");
                t1 = new Thread(g1);
                t2 = new Thread(g1);
```

Another Simple Example

```
            t1.start();    t2.start();
            for(int i=0; i<20; i++)
            {
                  System.out.println("MARKET,
                   + "cur thrd is : "
                   + Thread.currentThread().
                  getName());
                  try {Thread.sleep(100);}
                  catch(InterruptedException e){}
            }
      }
}
```

Another Simple Example

```
class Greengrocer implements Runnable
{
      private String product;
      private int iInstance = 0;
      private int iStatic = 0;
      Greengrocer(String str)
      {
            product = str;
      }
      public void run()
      {
            for(int iLocal=0; iLocal<20; iLocal++)
            {
```

Another Simple Example

```
                    System.out.println(product
                           +", cur thrd is : "
                    + Thread.currentThread().getName()
                    + " iLocal="+iLocal+" Instance="
                           +(iInstance++)
                    + " iStatic="+(iStatic++));
                    try {Thread.sleep(100);}
                    catch(InterruptedException e) {}
            }
        }
    }
```

Threads & Applets

- The main thread in each applet is the thread that calls the methods: init(), start(), stop() and destroy().
- It is very common creating another thread that treats multimedia as well as other tasks of the thread.
- The following example presents an applet that works as a digital clock.

Threads & Applets

```
import java.awt.*;
import java.applet.*;
import java.util.*;
public class DigiApplet extends Applet implements Runnable
{
    private Thread thread;
    private boolean goOn = true;
    private Date time = new Date();
```

Threads & Applets

```
public void start()
{
        goOn = true;
        if(thread==null)
        {
                thread = new Thread(this);
                thread.start();
        }
}
```

Threads & Applets

```
public void paint(Graphics g)
{
        g.drawString(String.valueOf(time),50,50);
}
public void run()
{
        while(goOn)
        {
                time = new Date();

```

Threads & Applets

```
                repaint();
                try
                {
                        Thread.sleep(1000);
                }
                catch(Exception e) {}
        }
}
```

Threads & Applets

```
    public void stop()
    {
        goOn=false;
        thread=null;
    }
}
```

Extending the class Thread

- Since the class Thread implements Runnable, it is possible extending the class and treat the new class instance both as the runnable object and both as the virtual CPU.

- When the new class is instantiated, the super class's constructor that is called is the one that doesn't have parameter\s.

Extending the Class Thread

```
public class SimpleThreadExtendDemo
{
    public static void main(String args[])
    {
            Greengrocer g1, g2;
            g1 = new Greengrocer("BANANA");
            g2 = new Greengrocer("TOMATO");
            g1.start();
            g2.start();
```

Extending the Class Thread

```
for(int i=0; i<20; i++)
{
        System.out.println(
        "MARKET, current thread is : "
        + Thread.currentThread().getName());
        try{Thread.sleep(100); }
        catch(InterruptedException e){}
}
```

Extending the Class Thread

```
        }
}
class Greengrocer extends Thread
{
        private String product;
        Greengrocer(String str)
        {
                product = str;
        }
```

Extending the Class Thread

```
        public void run()
        {
                for(int i=0; i<20; i++)
                {
                        System.out.println(product
                        +", current thread is : “+
                        Thread.currentThread());
                        try { Thread.sleep(100);}
                        catch(InterruptedException e) {}
                }
        }
}
```

Time Slicing & Preemptive

- When the scheduler is **preemptive** then many threads might be runnable but only one is actually running. The running thread continues in its running until one of the following happens:
  - It cease from being runnable
  - Another thread with an higher priority becomes runnable
  - It ends

Time Slicing & Preemptive

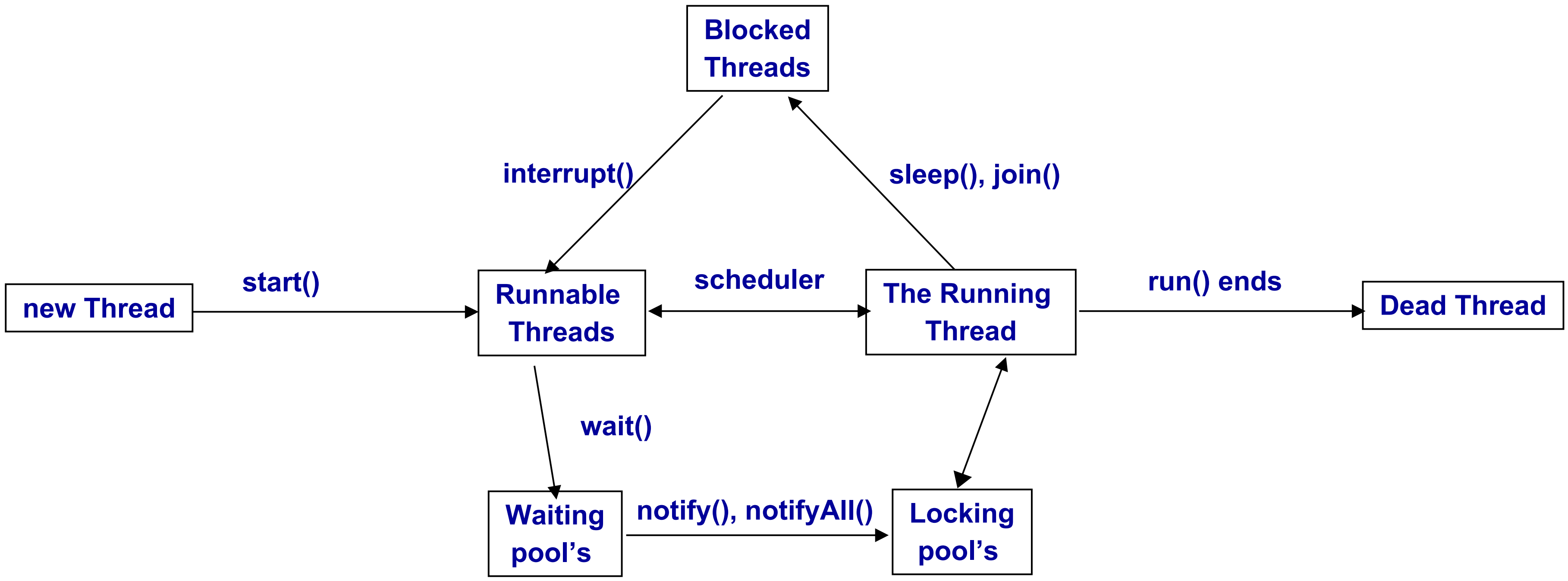
- When the scheduler is **time slicing** then many threads might be runnable and the one that is actually running might be preempted so other thread get the opportunity to run as well (even if their priority is lower). The JVM divides its time between all of the threads.

Priority Pools

- All runnable threads are kept in pools according to their priorities. When a thread becomes runnable it is placed in its appropriate runnable pool.
- When using the setPriority method use the static final variables that were declared in Thread.

Threads Scheduling

- The thread can be in each one of the following states:



Killing a Thread

- In order to kill a thread don’t call the stop() method. Instead, declare the class that implements Runnable in a way that enables calling a method which causes the termination of the run method.
- The following example presents two Greengrocers that sell their products and the first of them that reaches sales of 10000 kills the other one.

Killing a Thread – Example

```
public class KillingThreadDemo
{
    public static Greengrocer g1, g2;
    public static Thread t1, t2;
    public static void main(String args[])
    {
            g1 = new Greengrocer("BANANA");
            g2 = new Greengrocer("TOMATO");
```

Killing a Thread – Example

```
            t1 = new Thread(g1);
            t2 = new Thread(g2);
            t1.start();
            t2.start();
        }
    }
```

Killing a Thread – Example

```
class Greengrocer implements Runnable
{
    private String product;
    private int iInstance = 0;
    private int iStatic = 0;
    private double sales = 0;
    private boolean goOn = true;
```

Killing a Thread – Example

```
Greengrocer(String str)
{
        product = str;
}
public void stopRunning()
{
        goOn = false;
}
```

Killing a Thread – Example

```
public void run()
{
        double sum = 0;
        for(int iLocal=0; iLocal<20 & goOn; iLocal++)
        {
                sales += (sum=2000*Math.random());
                System.out.println(product
                        +" was sold in " + sum +
                        " the total sales are : " + sales);
                if(sales>=10000)
                {
```

Killing a Thread – Example

```
                 ((Thread.currentThread()==KillingThreadDemo.t1)?
                   KillingThreadDemo.g2:KillingThreadDemo.g1).
                  stopRunning();
                 }
                 try  { Thread.sleep(100); }
                 catch(InterruptedException e) {}
         }
     }
 }
```

The join() Method

- Causes the current thread to wait until the thread on which it is called terminates.

- The join() method has an overloaded version that receives in seconds a timeout value.

The join() Method - Example

```
public class JoinDemo
{
    public static Greengrocer g1, g2;
    public static Thread t1, t2;

    public static void main(String args[])
    {
            g1 = new Greengrocer("BANANA",10);
            g2 = new Greengrocer("TOMATO",20);
```

The join() Method - Example

```
t1 = new Thread(g1);
t2 = new Thread(g2);
t1.start();
t2.start();
System.out.println("I am the market manager.”
+” Now I shell wait until t1 is finished");
try {t1.join();} catch(InterruptedException e) {}
System.out.println("I am the market manager.”
+” Now I shell wait until t2 is finished");
try {t2.join();} catch(InterruptedException e) {}
```

The join() Method - Example

```
                System.out.println("Now that both t1 and t2 "
                +" are finished I can continue ...");
        }
}

class Greengrocer implements Runnable
{
            . . .
}
```

Synchronization - Intro

- Sometimes, different threads share the same data. One example for that can be a Stack instance that more than one thread share.

- Shared data might be problematic. The following is a possible Stack class declaration.

Synchronization - Intro

```
class Stack
{
    int index=0;
    int vec[] = new int[100];
    void push(int num)     { vec[index] = num; index++; }
    void pop()             { index--; return vec[index]; }
}
```

Synchronization - Intro

- That was just one sample for the problems that arise when multiple threads accessing shared data.

- There is a need in a mechanism that ensures the shared data is in a consistent state before any thread starts using it.

Synchronization – The ‘Lock Flag’

- Every object has within it a “lock flag” variable mechanism.
- This “lock flag” mechanism is extended from class Object.
- This “lock flag” mechanism is used in synchronizing between different thread.

Synchronization – The 'Lock Flag'

- The way of using the synchronized block is as follows:

```
synchronized( reference )
{
        …
}
```

Synchronized Method

- A method that all of its code is wrapped in a synchronized block that synchronizes to the object on which the methods works (this) can be marked as a synchronized method instead of having a synchronized block within it.

Synchronized Method

- The following two code fragments are equivalent:

```
void doSomething()
{
        synchronize(this)
        {
        …
        }
}
```

```
void synchronized doSomething()
{
        …


}
```

Deadlocks

- When multiple threads compete for the same lock flags, a thread might find itself waiting for a lock-flag that will never be available. This situation is known as “Deadlock”.

- It is the programmer responsibility avoiding the deadlocks situations.

`wait()` and `notify()`

- Sometimes, different threads that perform unrelated tasks and share their data need to have a way to interact with each other ensuring the shared data is kept protected from being corrupted.
- The wait() and `notify()` methods were declared in class Object.

`wait()` and `notify()`

- If during a thread execution, the `wait()` method is called on a rendezvous object, that thread pauses its running. The thread is moved to the waiting pool of that rendezvous object.

- If during a thread execution, the `notify()` method is invoked on a rendezvous object, threads that reside in that object’s waiting pool are moved to the locking pool (of the same specific object).

The wait() and notify() Example

- The following example presents a possible scenario for using the `wait()` and `notify()` methods.

- The example presents two threads: one adds Website objects to a given collection and the other prints each Website’s data to the screen.

The wait() and notify() Example

```
import java.util.*;
public class WaitNotifyExample
{
        public static void main(String argsp[])
        {
                String vec[] = {
                        "www.abrakadabra.com", "www.jumpjava.com",
                        "www.formidable.com", "www.samiandsusu.com",
                        "www.internick.com", "www.formula.com",
                        "www.solomon.com", "www.falafel.com",
                        "www.toledano", "www.davidka.com"};
```

The wait() and notify() Example

```
            WebsitesStack stack = new WebsitesStack();
            Thread t1, t2;
            WebsitesRobot robot =
                new WebsitesRobot(vec,stack);
            WebsitesReporter reporter =
                New WebsitesReporter(stack,vec.length);
            t1 = new Thread(robot);
            t2 = new Thread(reporter);
            t1.start();
            t2.start();
        }
    }
```

The wait() and notify() Example

```
class Website
{
        private String url;
        Website(String str)    { url = str; }
        void check()
        {
                try { Thread.sleep((int)(2000*Math.random())); }
                catch(InterruptedException e) {}
        }
        public String toString()      {  return url; }
}
```

The wait() and notify() Example

```
class WebsitesReporter implements Runnable
{
        WebsitesStack stack;
        int numOfSites;
        WebsitesReporter(WebsitesStack stack, int numOfSites)
        {
                this.stack = stack;
                this.numOfSites = numOfSites;
        }
```

The wait() and notify() Example

```
        public void run()
        {
                for(int i=0; i<numOfSites; i++)
                {
                        Website current = stack.pop();
                        try{ Thread.sleep((int)(2500*Math.random()));}
                        catch(InterruptedException e){}
                        System.out.println("["+Thread.currentThread().
                        GetName()+"]"+current);
                }
        }
}
```

The wait() and notify() Example

```
class WebsitesRobot implements Runnable
{
        WebsitesStack stack;
        String urls[];
        WebsitesRobot(String urls[], WebsitesStack stack)
        {
                this.stack = stack;
                this.urls = urls;
        }
```

The wait() and notify() Example

```
    public void run()
    {
        Website current;
        for(int i=0; i<urls.length; i++)
        {
            current = new Website(urls[i]);
            current.check();
            stack.push(current);
        }
    }
}
```

The wait() and notify() Example

```
class WebsitesStack
{
        private int index;
        private Website vec[] = new Website[100];
        synchronized void push(Website website)
        {
                notify();
                System.out.println("["
        +Thread.currentThread().getName()+"] notify...");
                vec[index] = website;
                index++;
```

The wait() and notify() Example

```
        System.out.println("["
        +Thread.currentThread().getName()
        +"] push() succeeded...");
}
synchronized Website pop()
{
        if(index==0)
        {
                try
                {
```

The wait() and notify() Example

```
                wait();
            }
            catch(InterruptedException exception)  {}
        }
        index--;
        return vec[index];
    }
}
```

Daemon Threads & User Threads

- Threads in Java can be marked as either user threads or as daemon threads.

- A new thread inherits this characteristic from the thread that invoked its constructor.

- The difference between the two is that the JVM exits and stops the entire program when all user threads are dead.

ThreadDump

- ThreadDump is a textual representation of currently executed threads.
- Using a ThreadDump it is possible to identify problems in our code such as deadlocks.
- Various IDEs include different types of ThreadDump tools through which we can get a detailed textual representation of the current running threads.

ThreadDump

- The simplest tool to use in order to get a ThreadDump is the “jstack” command tool.
- The simplest way to run this tool is by providing it with the process id for which we want to get the
  c:\> jstack -l [process id]
- The simplest way to get all process id numbers is to use the jps tool
  c:\> jps

C:\WINDOWS\system32\cmd.exe

```
C:\>jps
4976 Jps
4960 StudentsDBServer
2456
4600 NanoHTTPD
1932 RegistryImpl

C:\>jstack 4600
2008-09-03 13:21:22
Full thread dump Java HotSpot(TM) Client VM (10.0-b19 mixed mode, sharing):

"Thread-0" daemon prio=6 tid=0x02f09400 nid=0x1488 runnable [0x0308f000..0x0308fd94]
   java.lang.Thread.State: RUNNABLE
        at java.net.PlainSocketImpl.socketAccept(Native Method)
        at java.net.PlainSocketImpl.accept(PlainSocketImpl.java:384)
        - locked <0x229ebbd0> (a java.net.SocksSocketImpl)
        at java.net.ServerSocket.implAccept(ServerSocket.java:453)
        at java.net.ServerSocket.accept(ServerSocket.java:421)
        at NanoHTTPD$1.run(NanoHTTPD.java:187)
        at java.lang.Thread.run(Thread.java:619)

"Low Memory Detector" daemon prio=6 tid=0x02a5dc00 nid=0x1508 runnable [0x00000000..0x00000000]
   java.lang.Thread.State: RUNNABLE

"CompilerThread0" daemon prio=10 tid=0x02a58000 nid=0x1080 waiting on condition [0x00000000..0x02d0f740]
   java.lang.Thread.State: RUNNABLE

"Attach Listener" daemon prio=10 tid=0x02a56c00 nid=0x1bc waiting on condition [0x00000000..0x00000000]
   java.lang.Thread.State: RUNNABLE

"Signal Dispatcher" daemon prio=10 tid=0x02a55c00 nid=0xd88 runnable [0x00000000..0x00000000]
   java.lang.Thread.State: RUNNABLE

"Finalizer" daemon prio=8 tid=0x02a4e000 nid=0x10f8 in Object.wait() [0x02c1f000..0x02c1fc94]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x22960b28> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:116)
        - locked <0x22960b28> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:132)
        at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:159)

"Reference Handler" daemon prio=10 tid=0x02a4d000 nid=0x10dc in Object.wait() [0x02bcf000..0x02bcfd14]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x22960a30> (a java.lang.ref.Reference$Lock)
        at java.lang.Object.wait(Object.java:485)
        at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:116)
        - locked <0x22960a30> (a java.lang.ref.Reference$Lock)

"main" prio=6 tid=0x00295400 nid=0x6d8 runnable [0x0090f000..0x0090fe54]
   java.lang.Thread.State: RUNNABLE
        at java.io.FileInputStream.readBytes(Native Method)
        at java.io.FileInputStream.read(FileInputStream.java:199)
        at java.io.BufferedInputStream.fill(BufferedInputStream.java:218)
        at java.io.BufferedInputStream.read(BufferedInputStream.java:237)
```

start | Hell... | 3 W... | thre... | 2 I... | Busi... | eMul... | 4 T... | 4 W... | Unti... | Star... | 1:21 PM

Blocking Queues

- The `java.util.concurrent.BlockingQueue` interface defines a queue we can use when implementing the “Producer Consumer” design pattern.

  The producer thread adds elements. The consumer thread retrieves them. The BlockingQueue object allows us handing over elements from one thread to another in a safe way. While having one (and only) thread that is capable of accessing the sensitive data (e.g. the consumer thread is the one and only thread that can access a collection of news items held within the blocking queue object) we can avoid using the synchronization mechanism.

Blocking Queues

- The BlockingQueue main methods include the following:

  `public boolean add(E e)`

  This method adds an element... and throws IllegalException if queue is full.

  `public boolean offer(E e)`

  This method adds an element and returns true. If queue is full returns false.

  `public void put(E e) throws InterruptedException`

  This method adds an element... if queue is full it blocks.

  `public E element()`

  This method returns the head element... it throws NoSuchElementException if queue is empty.

Blocking Queues

`public E peek()`

This method returns the head element... and returns null if the queue is empty.

`public E poll()`

This method removes and returns the head element... and returns null if the queue is empty.

`public E remove()`

This method removes and returns the head element... if the queue is empty it throws NoSuchElementException.

`public E take()`

This method removes and returns the head element... if the queue is empty it blocks.

Blocking Queues

- The `java.util.concurrent` package includes several implementations of the BlockingQueue interface:

  `LinkedBlockingQueue`

  This blocking queue implementation can be used without any capacity limit.

  `ArrayBlockingQueue`

  This blocking queue implementation does have a limit on its capacity.

Blocking Queues

`PriorityBlockingQueue`

This blocking queue implementation doesn't behave as a conventional queue. It is a priority queue that that allows removing its elements in accordance with their priority.

`DelayQueue`

This blocking queue allows removing its elements when their delay ends only.

Blocking Queues

```
import java.util.concurrent.*;

public class ConsumerProducerDemo
{
        public static void main(String[] args)
        {
                BlockingQueue<String> queue =
                        new LinkedBlockingQueue<String>(6);
                Runnable producer =
                        new Producer("Producer", queue);
                Runnable consumer =
                        new Consumer("Consumer", queue);
                Thread t1 = new Thread(producer);
                Thread t2 = new Thread(consumer);
                t1.start();
```

Blocking Queues

```
                try
                {
                        Thread.sleep(10000);
                }
                catch (InterruptedException e)
                {
                        e.printStackTrace();
                }
                t2.start();
        }
}
```

Blocking Queues

```
import java.util.concurrent.*;

class Producer implements Runnable
{
        private String name;
        private BlockingQueue<String> queue;
        private String vec[] = {
"Chris","Santa","Gil","Doron","Omer","Haim","Tamar","George","
Angela","Anna","Steve","Tom","Greg","Michael", "David",
"John", "Jane", "Sam", "Karl", "Fred" };

        public Producer(String name,
                BlockingQueue<String> queue)
        {
                this.name = name;
                this.queue = queue;
        }
```

Blocking Queues

```
    public synchronized void run()
    {
        for (int i = 0; i < vec.length; i++)
        {
            try
            {
                queue.put(vec[i]);
                System.out.println(name + " puts " + vec[i]);
                Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

Blocking Queues

```
import java.util.concurrent.*;

class Consumer implements Runnable
{
        private String name;
        private BlockingQueue<String> queue;

        public Consumer(String name, BlockingQueue<String> queue)
        {
                this.name = name;
                this.queue = queue;
        }
```

Blocking Queues

```
    public synchronized void run()
    {
        for (int i = 0; i < 20; i++)
        {
            try
            {
                String str = queue.take();
                System.out.println(name + " takes " + str);
                    Thread.sleep(200);
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

Thread Safe Collections

- The `java.util.concurrent` package includes efficient implementations for various collection classes, such as maps, sorted sets and queues:

```
ConcurrentHashMap
ConcurrentSkipListMap
ConcurrentSet
ConcurrentLinedQueue
...
```

Thread Safe Collections

- Those thread safe implementations use sophisticated algorithms that minimize threads contention.
- The iterators these collections return is a weakly consistent one. The iterators may (or may not) reflect all modifications made after they were constructed.

Callables & Futures

- The `Callable` parameterized interface includes one method only.

```
public interface Callable<V>
{
     V call() throws Exception;
}
```

- `Callable` is a parameterized type. The parameterized type is the type of the returned value we get when we call the method call().

Callables & Futures

- A `Callable` object represents an asynchronous computation that eventually returns a value of the parameterized type.

Callables & Futures

- The `Future` parameterized interface includes five methods:

```
public interface Future<E>
{
        V get() throws
                InterruptedException,
                ExecutionException;
        V get(long timeout, TimeUnit unit) throws
                InterruptedException,
                ExecutionException,
                TimeoutException;
        void cancel(boolean mayInterrupt);
        boolean isCancelled();
        boolean isDone();
}
```

Callables & Futures

- The `Future` object holds the result of an asynchronous computation represented by a Callable object.
- The `FutureTask` parameterized wrapper class implements `Future` and `Runnable`. The `FutureTask` constructor receives a reference for a `Callable` object. A `FutureTask` object can be the `Runnable` object connected with a `Thread` object.

Callables & Futures

```
Callable<Integer> myComputation = . . .;

FutureTask<Integer> task =
        new FutureTask<Integer>(myComputation);

Thread t = new Thread(task);

t.start();

. . .

Integer result = task.get();
```

Callables & Futures

- Calling `get()` on our `Future` object blocks until we get the result. Calling `get(long,TimeUnit)` on our `Future` object blocks either till we get the result or a `TimeoutException` is thrown.

Callables & Futures

```
import java.util.concurrent.*;


public class FutureTaskDemo
{
  public static void main(String args[])
  {
    Callable<Double> myComputation = new MyComputation();
    FutureTask<Double> task =
        new FutureTask<Double>(myComputation);
    Thread t = new Thread(task);
    t.start();
    System.out.println("continue with our program...");
    System.out.println("continue with our program...");
    System.out.println("continue with our program...");
    System.out.println("calling task.get()...");
    Double result = null;
```

Callables & Futures

```
                try
                {
                        result = task.get();
                }
                catch(InterruptedException e)
                {
                        e.printStackTrace();
                }
                catch(ExecutionException e)
                {
                        e.printStackTrace();
                }
                System.out.println("task.get() returns...");
                System.out.println("result="+result);
        }
}
```

Callables & Futures

```
class MyComputation implements Callable<Double>
{
        public Double call() throws Exception
        {       double averages[] = new double[40];
                for(int m=0; m<40; m++) {
                        double total = 0;
                        for(int i=0; i<1000000; i++) {
                                total += Math.random();
                        }
                        averages[m] = total / 1000000;
                }
                double sum = 0;
                for(int k=0; k<40; k++)
                {
                        sum += averages[k];
                }
                return sum / 40;
        }
}
}
```

Executors

- Creating new threads is an expensive task. The more threads we create during our program execution the more resources we need from our platform.
- Instead of creating a new thread each time we need one we better use a pool of idle threads ready to be used... a pool to which we can return a thread once we completed to use it.

Executors

- The `Executors` class includes several static factory methods we can use to construct new thread pools.

  `public static ExecutorService newFixedThreadPool(int num)`

  The returned pool includes a fixed set of threads. The idle threads are kept indefinitely.

  `public static ExecutorService newSingleThreadExecutor()`

  The returned pool includes a single thread that executes the submitted tasks sequentially.

  `...`

Executors

- Each one of the threads pools is of type `ExecutorService`. One of the methods this interface defines is `submit`.
- Its overloaded versions include the following:

```
Future<T> submit(Callable<T> task);
Future<T> submit(Runnable task, T result);
Future<?> submit(Runnable task);
```

Executors

- When we complete using our thread pool we can call the `shutdown()` method in order to initiate a shutdown sequence for the pool.

Executors

```
import java.util.concurrent.*;

public class ExecutorsDemo
{
  public static void main(String args[])
  {
    ExecutorService pool = Executors.newFixedThreadPool(3);
    PrintJob []vec = { new PrintJob("david",4),
        new PrintJob("michael",12),
        new PrintJob("john",2),
        new PrintJob("angela",6),
        new PrintJob("anna",2),
        new PrintJob("debby",1)};
    for(int i=0; i<vec.length; i++)
    {
        pool.submit(vec[i]);
    }
  }
}
```

Executors

```
public class PrintJob implements Runnable
{
        private String name;
        private int length;
        public PrintJob(String str, int length)
        {
                this.name = str;
                this.length = length;
        }
        public void run()
        {
                System.out.println(name+" started");
                for(int i=0; i<length; i++)
                {
                        try{Thread.sleep(1000);}
                        catch(Exception e){}
                }
                System.out.println(name+" completed");
        }
}
```

Synchronizers Classes

- The `java.util.concurrent` package contains several predefined classes that assist managing sets of collaborating threads in various common scenarios.

CyclicBarrier

- When having a set of threads that a predefined number of them first need to complete or to reach a specific point before we can move forward in our program using their outcome.

CyclicBarrier

```
import java.util.concurrent.*;

public class CyclicBarrierDemo
{
        private static char matrix[][] =
                {
                        { 'h', 'i' },
                        { 'a', 'b', 'b', 'a' },
                        { 'm', 'a', 'm', 'a' },
                        { 'j', 'a', 'v', 'a', 'f', 'x' },
                        { 'i', 's', 'r', 'a', 'e', 'l' } };
        private static String results[] =
                new String[matrix.length];
```

CyclicBarrier

```
private static class StringCreator extends Thread
{
        int row;

        CyclicBarrier barrier;

        StringCreator(CyclicBarrier barrier, int row)
        {
                this.barrier = barrier;
                this.row = row;
        }
```

CyclicBarrier

```
public void run()
{
        int columns = matrix[row].length;
        String str = "";
        for (int i = 0; i < columns; i++)
        {
                try{Thread.sleep(500);}
                catch (InterruptedException e)
                {
                        e.printStackTrace();
                }
                str += matrix[row][i];
        }
        results[row] = str;
        System.out.println("Result for row " + row
                + " is : " + str);
```

CyclicBarrier

```
        // wait for others
        try
        {
                barrier.await();
        }
        catch (InterruptedException ex)
        {
                ex.printStackTrace();
        }
        catch (BrokenBarrierException ex)
        {
                ex.printStackTrace();
        }
    }
}
```

CyclicBarrier

```
public static void main(String args[])
{
  Runnable merger = new Runnable()
  {
    public void run()
    {
      String str = "";
      for (int i = 0; i < matrix.length; i++)
      {
        str += "\n"+results[i];
      }
      System.out.println("Results is: " + str);
    }
  };
```

CyclicBarrier

```
    CyclicBarrier barrier = new CyclicBarrier(matrix.length, merger);
    for (int i = 0; i < matrix.length; i++)
    {
      new StringCreator(barrier, i).start();
    }
    System.out.println("Waiting...");
  }
}
```

Exchanger

- When having two threads working each one of them on object of the same type. The first thread is filling the first object and the second thread is emptying it. The Exchanger allows the two threads to exchange the object when both of them are ready for doing so.

Exchanger

```
public class Ball
{
        private int id;
        public Ball()
        {
                int num = 0;
                for(int i=0; i<4; i++)
                {
                        num = 10*num + (int)(10*Math.random());
                }
                id = num;
        }
        public String toString()
        {
                return String.valueOf(id);
        }
}
```

Exchanger

```
public class Bucket
{
        private Ball balls[] = new Ball[10];
        private int numOfBalls = 0;
        private int id;

        public Bucket()
        {
                int num = 0;
                for(int i=0; i<2; i++)
                {
                        num = 10*num + (int)(10*Math.random());
                }
                id = num;
        }
```

Exchanger

```
public int getNumOfBalls()
{
        return numOfBalls;
}

public synchronized void addBall(Ball ob)
{
        if(numOfBalls<10)
        {
                balls[numOfBalls++] = ob;
        }
        try
        {
                Thread.sleep(200);
        }
        catch(InterruptedException e)
        {
                e.printStackTrace();
        }
}
```

Exchanger

```
public synchronized Ball getBall()
{
        Ball ob = null;
        if (numOfBalls > 0)
        {
                ob = balls[--numOfBalls];
                if((numOfBalls+1)<10)
                        balls[numOfBalls+1] = null;
        }
        try
        {
                Thread.sleep(200);
        }
        catch(InterruptedException e)
        {
                e.printStackTrace();
        }
        return ob;
}
```

Exchanger

```
    public String toString()
    {
            String str = "bucket id="+id+" ";
            for(int i=0; i<numOfBalls; i++)
                    str += balls[i].toString()+" ";
            return str;
    }

}
```

Exchanger

```
import java.util.concurrent.*;

public class ExchangerDemo
{
        private static Exchanger<Bucket> exchanger =
                new Exchanger<Bucket>();

        public static void main(String args[])
        {
                Thread t1 = new BallsProducer();
                t1.start();
                Thread t2 = new BallsConsumer();
                t2.start();
        }
```

Exchanger

```
public static class BallsProducer extends Thread
{
  Bucket bucket;

  public BallsProducer()
  {
    bucket = new Bucket();
    System.out.println("producer creates "+bucket);
  }
```

Exchanger

```
public void run()
{
  try
  {
    while (true)
    {
      while(bucket.getNumOfBalls()<10)
      {
        Ball ball = new Ball();
        bucket.addBall(ball);
        System.out.println("===> producer adds "+ball);
      }
      System.out.println("producer gives "+bucket);
      bucket = exchanger.exchange(bucket);
      System.out.println("producer receives "+bucket);
    }
  } catch (InterruptedException e){e.printStackTrace();}
}
}
```

Exchanger

```
public static class BallsConsumer extends Thread
{
        Bucket bucket;
        public BallsConsumer()
        {
                bucket = new Bucket();
                System.out.println("consumer creates "+bucket);
        }
```

Exchanger

```
    public void run()
    {
      try
      {
        while (true)
        {
          System.out.println("consumer gives "+bucket);
          bucket = exchanger.exchange(bucket);
          System.out.println("consumer receives "+bucket);
          //emptying bucket
          while(bucket.getNumOfBalls()>0)
          {
          System.out.println("===> consumer get "+
               bucket.getBall());
          }
        }
      } catch (InterruptedException e) { e.printStackTrace();}
    }
  }
}
```

CountDownLatch

- When having one (or more) threads that need to wait until a specified number of events have occurred. This class allows us to have all threads in wait till the decremented counter reaches 0.

CountDownLatch

```
import java.util.concurrent.*;


public class CountDownLatchDemo
{
  private static CountDownLatch counter;
  public static void main(String args[])
  {
    counter = new CountDownLatch(3);
    Worker vec[] = {new Worker("david"), new Worker("michael")};
    vec[0].start();
    vec[1].start();
    for(int i=0; i<10; i++)
    {
      //do something
      try{Thread.sleep(2000);}catch(InterruptedException e)
      {e.printStackTrace();}
      System.out.println("do something...");
      counter.countDown();
    }
  }
```

CountDownLatch

```
  static class Worker extends Thread
  {
    String name;
    Worker(String name) {this.name = name;}
    public void run()
    {
      try{counter.await();}catch(InterruptedException e)
      {e.printStackTrace();}
      for(int i=0; i<10; i++)
      {
        System.out.println(this.name+" worker");
        try{Thread.sleep(100);} catch(InterruptedException e)
        {e.printStackTrace();}
      }
    }
  }
}
```

Semaphore

- When having a case in which we want to restrict the access to specific object to a predefined number of threads that can access it at the same time. If the predefined number of threads is 1 then at any point of time only one thread will be able to access our specific object.

Semaphore

```
import java.util.concurrent.*;


public class SemaphoreDemo
{
   private static Semaphore locker;
   public static void main(String args[])
   {
        locker = new Semaphore(2);
        String names[] = {
                "david",
                "michael",
                "doron",
                "mike",
                "nataly",
                "taly",
                "steve",
                "ricky",
                "bruce",
                "karl",
                "angela"
        };
```

Semaphore

```
    for(int i=0; i<names.length; i++)
    {
       new Worker(names[i],(int)(40*Math.random())).start();
    }
}

static class Worker extends Thread
{
    String name;
    int num;
    Worker(String name,int num)
    {
        this.name = name;
        this.num = num;
    }
```

Semaphore

```
        public void run()
        {
            try
            {
                locker.acquire();
                for(int i=0; i<num; i++)
                {
                    System.out.println(this.name+" worker");
                    Thread.sleep(50);
                }
                locker.release();
            }
            catch(InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

Locks

- The `Lock` interface describes an object we can use to lock the access into a specific code segment and limit it to one thread at a time, which will be the one who holds the lock.
- The `ReentrantLock` class is one of the available classes that implements `Lock`.

Locks

```
ReentrantLock locker = new ReentrantLock();
...
locker.lock();
try
{
     critical section
}
finally
{
     locker.unlock();
}
```

Locks

- Calling the `newCondition()` method on our Lock object returns an object of type `Condition`.
- When a required condition isn't true, the executing thread can call the `await()` method on the `condition` object. As a result of that, the lock will be released.

Locks

- When another thread causes the required condition to be true it can call the `signal()` or the `signalAll()` method on the relevant `condition` object. That will cause one of the waiting threads or all waiting threads to try and acquire back the lock they gave up due to not meeting the condition and resume their execution.

Locks

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class MyStack
{
  private Integer vec[] = new Integer[5];
  private int counter = 0;
  private Lock lock = new ReentrantLock();
  private Condition isNotFull = lock.newCondition();
  private Condition isNotEmpty = lock.newCondition();

  public Integer pop() throws InterruptedException
  {
    lock.lock();
```

Locks

```
  try
  {
        while(counter==0) //empty
        {
          isNotEmpty.await();
        }
        Integer ob = vec[--counter];
        vec[counter] = null;
        isNotFull.signal();
        System.out.println(ob+" was received and stack has "+counter
          +"elements");
        return ob;
  }
  finally
  {
        lock.unlock();
  }
}
```

Locks

```
  public void put(Integer element) throws InterruptedException
  {
    lock.lock();
    try
    {
      while(counter==vec.length) { isNotFull.await(); }
      vec[counter] = element;
      counter++;
      isNotEmpty.signal();
      System.out.println(element+" was added and stack has "+counter
        +" elements");
    }
    Finally { lock.unlock(); }
  }
}
```

Locks

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class MyStackDemo
{
  public static void main(String args[])
  {
    final MyStack stack = new MyStack();
```

Locks

```
new Thread()
{
  public void run()
  {
    for (int i = 1; i < 100; i++)
    {
      try
      {
        Thread.sleep((int) (1000 * Math.random()));
        int num = (int)(200 * Math.random());
        stack.put(num);
      }
      catch (InterruptedException e)
      {
        e.printStackTrace();
      }
    }
  }
}.start();
```

Locks

```
    new Thread()
    {
      public void run()
      {
        for (int i = 1; i < 100; i++)
        {
          try
          {
            Thread.sleep((int) (1000 * Math.random()));
            int num = stack.pop();
          }
          catch (InterruptedException e)
          {
            e.printStackTrace();
          }
        }
      }
    }.start();
  }
}
```

The Fork/Join Framework

- As of Java 7 we can use this framework in order to instruct on the execution of computation work into smaller pieces and have those pieces executed on separated cores.

The `ForkJoinPool` Class

- This class implements the ExecutorService interface.
- Unlike most other ExecutorService implementations this class employ a work-stealing algorithm.
- All threads attempt to find and execute subtasks created by other active tasks. Worker threads that run out of things to do can steal tasks from other threads that are still busy.

The `ForkJoinTask` Class

- We should define a new class that extends this class or one of its two specialized types `RecursiveTask` or `RecursiveAction`.
- The object instantiated from the new defined class will represent the computation work we want to perform on multiple processors.

The `ForkJoinTask` Class

- We should implement the `compute()` function in order to perform the work directly (if it is not a big segment) or split it into pieces (When it is a big segment).

```
if(segment is small enough)
    do it directly
else
    split the segment into smaller segments
```

The `ForkJoinTask` Class

```java
package com.abelski.samples;

import java.util.Vector;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

public class ForkJoinDemo
{
	public static void main(String[] args)
	{
		System.out.println("available processors: "+
			Runtime.getRuntime().availableProcessors());
		ForkJoinPool pool = new ForkJoinPool();
		pool.submit(new ComputationWork(0,1200));
		//...
	}
}
```

The `ForkJoinTask` Class

```
class ComputationWork extends RecursiveAction
{
        int to;
        int from;

        public ComputationWork(int from,int to)
        {
                super();
                this.to = to;
                this.from = from;
        }

        private static final long serialVersionUID = 1L;
```

The `ForkJoinTask` Class

```
@Override
protected void compute()
{
        if((to-from)<100)
        {
                computeDirectly();
        }
        else
        {
                int num = to-from;
                ComputationWork first =
                        new ComputationWork(from,num/2);
                ComputationWork second =
                        new ComputationWork(num/2,to);
                invokeAll(first,second);
        }
}
```

The `ForkJoinTask` Class

```
protected void computeDirectly()
{
        int i=from;
        while(i<=to)
        {
                //do something that takes time
                i++;
                }
        }
}
```