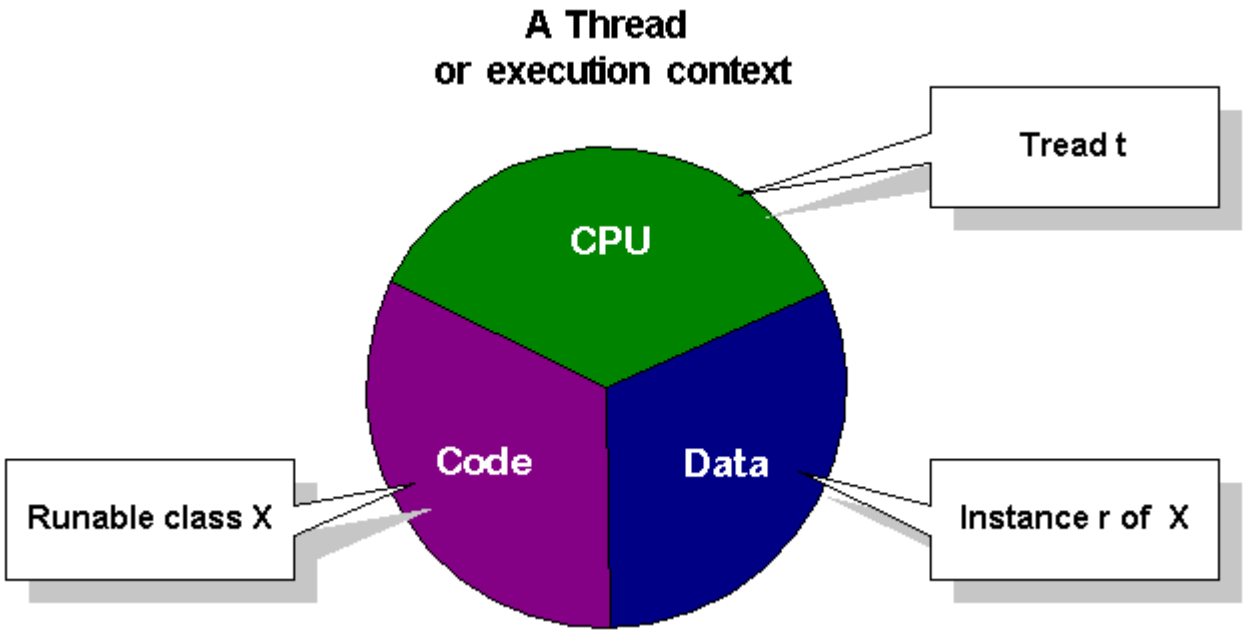
CST8221 – Java Application Programming
Unit 9
Introduction to Java Threads

What is a Thread?

The term *thread* is shorthand for thread of control, and a thread of control is, at its simplest, a section of code executed independently of other treads of control within a single program. Single-threaded systems use an approach known as *event loop* or *polling*. This approach has many disadvantages. For example, in a single-threaded environment when a thread blocks because it is waiting for some resource, the entire program stops and waits. This wastes CPU time. The benefit of multithreading is that the main loop (polling) is eliminated. One thread can wait without stopping other parts of the program. The other threads can still run.
The Java runtime system (JVM) depends on threads for many things, and all the class libraries are designed with multithreading in mind. This allows the entire Java environment to be asynchronous. When a Java program is launched, one thread starts running immediately. This is the *main* thread of the program. This thread can spawn other threads. Java also supports *daemon* threads. The Java runtime will not exit until all normal threads stop running but if there are only *daemon* threads running, than the run time will exit. This is because the *daemon* threads are considered service provider (or servers).

In Java, a thread (or execution context) is the encapsulation of a virtual CPU with its own code and data. The Java threads are implemented in the class *java.lang.Thread*. This class allows you to create and control the behavior of the threads.



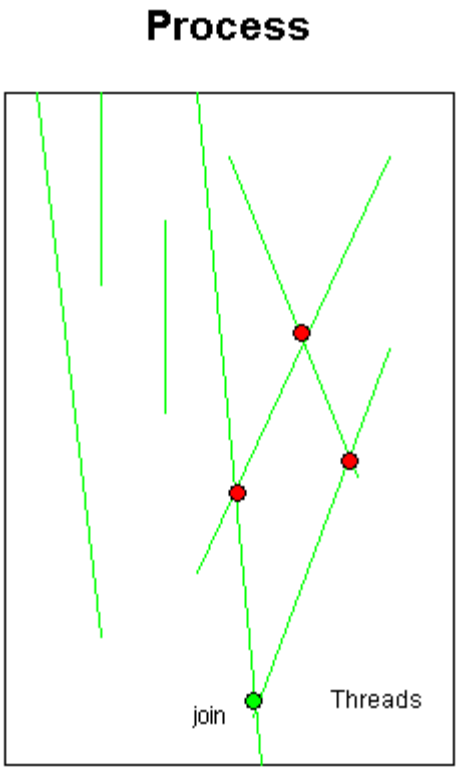
The Parts of a Thread

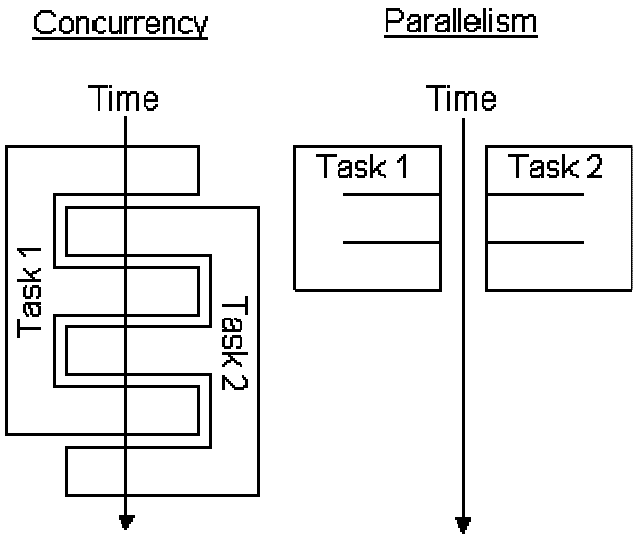
A thread in Java consist of three basic parts:

- A virtual CPU. In Java, the virtual CPU is encapsulated in an instance of the Java *Thread* class. When the thread is created, the code and the data that define its contents are determined by the object passed to the thread constructor.
- The code the CPU is executing. The code is provided by the methods of a class. The thread class can have many instances; that is, the code can be shared by multiple threads, independent of data.
- The data, which the code manipulates. Data may or may not be shared. Two or more threads share the same data when they share access to a common object.

Where does a Thread live?

A thread lives in a process. A process is program in execution. Threads share the process CPU time, memory, and resources.

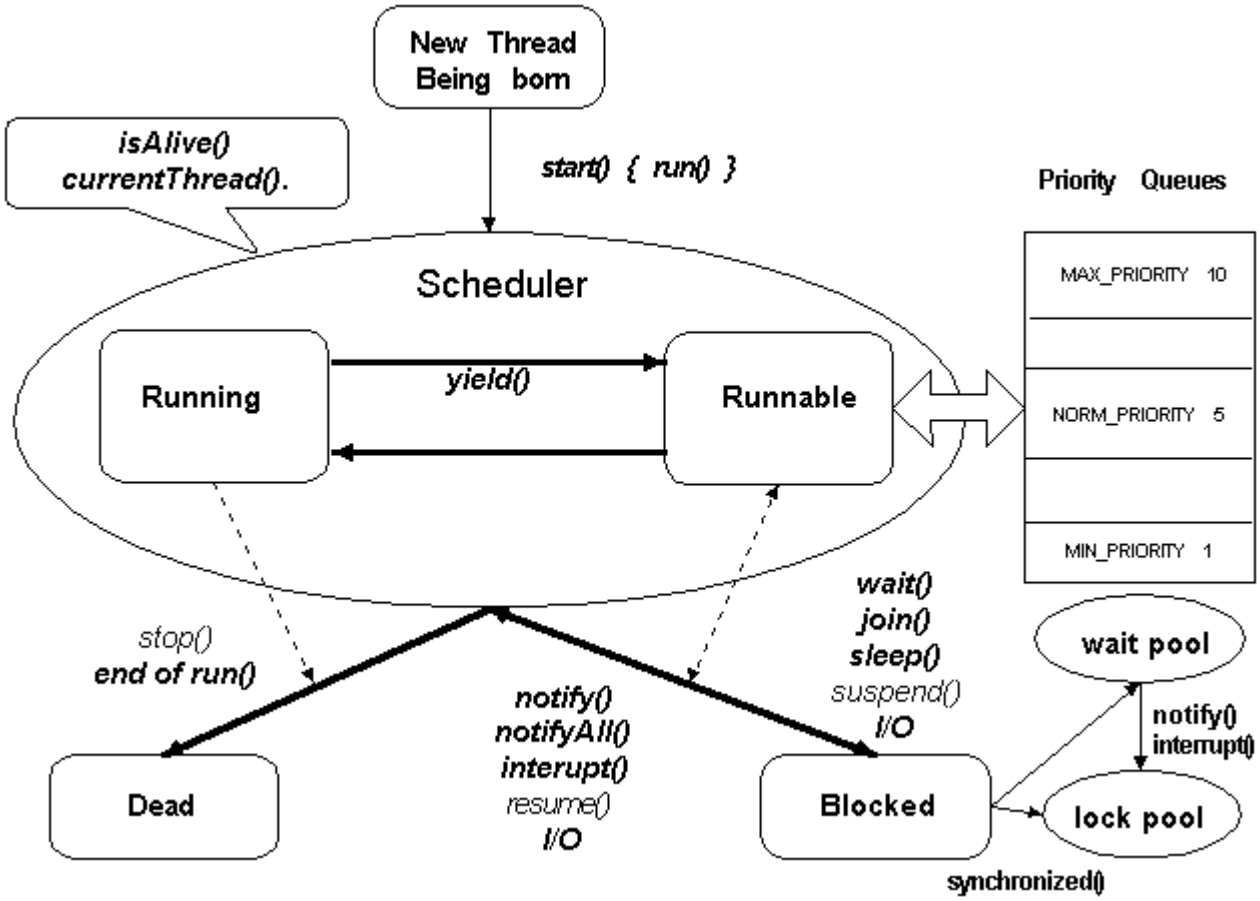




Threads run concurrently within a process.

The Life of a Thread

From the point of its creation to the point of its dead, a thread goes through and can exist in many different states. The complex life of a Java thread is illustrated in the figure below. The thread life is controlled by the programmer and the Java thread scheduler. In Java, threads are usually pre-emptive but not necessarily time sliced. The Java, the thread scheduling is based on priority.



Creating a Thread

In Java, a thread can be created in two different ways:

- Implementing the *Runnable* interface
- Extending the *Thread* class

Example of implementing the Runnable interface

```
public class CreateThread{
    public static void main(String [] args){
        Runnable r = new SomeRunnable();
        Thread t = new Thread(r);
        t.start();
    }
}

class SomeRunnable implements Runnable{
    int life;
    public void run(){
     life = 1;
     while (true){
      System.out.println("I am alive: " + life++);
            if(life == 111) break;
     }
    }
}
```

Example of extending the Thread class

```
public class SomeThread extends Thread{
    private int life;
    public void run{
     life = 1;
     while (true){
      System.out.println("I am alive: " + life++);
      try{
        sleep(100);
      }catch(InterruptedException e{}
            if(life == 111) break;
     }

    }
    public static void main(String [] args){
        Thread t = new SomeThread("MyThread");
        t.start();
    }
}
```

Which is the right way to create a thread?

- From object-oriented point of view, and because of the Java single inheritance, implementing the ***Runnable*** interface is the recommended approach for creating threads.
- When a run() method is overridden in a class that extends the Thread class, *this* refers to the actual Thread instance that is controlling execution. Therefore instead getting access to the current thread by way of the call

  ```
  Thread.currentThread().sleep();
  ```

  or

  ```
  Thread.sleep();
  ```

  it can be done through

  ```
  sleep(100);
  ```

Extending Thread results in a simpler code. This is the reason many programmers use this mechanism.

The **recommended approach** for creating a thread is to create a class implementing the ***Runnable*** interface.

Stopping a Thread

The *stop()* (and *suspend()*)has been deprecated in Java 1.2. It was found that they are inherently unsafe and can lead to data corruption (or deadlocks). It was concluded that the thread should run to its natural termination, that is, to the end of its *run()* method. Since most of the run() methods contain an endless loop, a special instance variable that acts as a flag should be used. In the tread the value of the variable should be checked frequently. If the thread waits for long periods, the *interrupt()* method can be called on the thread to interrupt the wait.

Example

```
//declaration of a class
//some members

private volatile boolean state = false;

//some methods

public void run(){
     try{
          while(!state){//do something}
     }catch(InterruptedException ie){
          return;
     }
}

public void stopRun(){
     state = true;
}
public void stop(){
     state = true;
}
```

Another Example

```
public class ControllableThread extends Thread{
     static final int SUSPEND = 1;
     static final int STOP = 2;
     static final int RUN = 1;
     private int status = RUN;

public synchronized void setStatus(int status){
    this.status = status;
    if(status == RUN) notify();
  }

  public synchronized boolean getStatus(){
     while (status == SUSPEND){
       try {
         wait();
       }catch (InterruptedException ie){}
     }
     if( status == STOP) return false;
     return true;
  }

 public void run(){
    while (true){
      System.out.println("Thread" +
           Thread.currentThread().getName() + "runs");
      if(!getStatus()){
        System.out.println("Thread"+
           Thread.currentThread().getName()+"stoped");
        break;
      }
    }
  }
}
```

Java Thread Synchronization

When threads are sharing data or resources serious problems can occur leading to data corruption or deadlocks.

The Problem

Imagine the class that represents a stack and two or more threads sharing it.

```
public class IntStack{
  int pointer = 0;
  int [] istack = new int[100];

  public void push (int data){
    istack[pointer] = data;
    ++pointer;
  }
  public int pop (){
    --pointer;
    return istack[pointer];

  }
}
```

Suppose thread Two is adding numbers and thread One is removing numbers. Thread Two just pushed a number, but has not yet incremented the stack pointer. At this point the thread is preempted.

istack |1|5|3| | |
pointer = 2

If thread Two resumes execution, there might be no damage, but suppose thread One is active and removes a number. It will pop 5:

istack |1|5|3 | | |
pointer = 1

Next, Two takes action and pushes another number. It will overwrite 3.

istack |1|5|3 | | |
pointer = 2

It is clear that the data model is inconsistent.

The Solution

If the thread can lock the shared data until it completes the operation, the corruption of the data will be prevented. Java provides such locking mechanism. Actually, in Java, every object has a lock associated with it. When a thread takes the lock of an object no other thread can operate on the locked data.

```
public class CookyStack
{
     private int index = 0;
     private char [] buffer = new char[6];

     public synchronized char pop() {
          while (index == 0) {
               try {
                    this.wait();
               } catch (InterruptedException e) {
                    // ignore it..
               }
          }
          this.notify();
          index--;
          return buffer[index];
     }

     public synchronized void push(char c) {
          while (index == buffer.length) {
               try {
                    this.wait();
               } catch (InterruptedException e) {
                    // ignore it..
               }
          }
          this.notify();
          buffer[index] = c;
          index++;
     }
}
```

Another Solution

```
import java.util.Vector;

public class CookyVJar {

    private Vector buffer;

    private CookyVJar (){
      buffer = new Vector(400, 200);
    }

    public static CookyVJar getCooky(){
      return new CookyVJar();
    }

    public synchronized char pop() {
    char c;

    while (buffer.size() == 0) {
        try {
        this.wait();
        } catch (InterruptedException e) {
        // ignore it...
        }
    }
          //this.notify(); // no waiting thread to notify
    c = ((Character)buffer.remove(buffer.size()-1)).charValue();
    return c;
    }

    public synchronized void push(char c) {

    this.notify();

    Character charObj = new Character(c);
    buffer.addElement(charObj);
    }
}
```

More Problems

One of the problems that can potentially occur when two ore more processes interact to perform a common task is *Deadlock*. Deadlock is a situation that occurs when a thread or process is prevented from running because it requires a resource that will never become available. *Starvation* is a similar process to Deadlock. Starvation occurs when a process is prevented from accessing a resource due to scheduling.

The following could create a Deadlock situation:

- **Mutual exclusion**. Only a single process can hold a resource or modify shared information at one time.
- **Circular waiting**. A thread A can wait for a thread B, which in turn waits on C, which in turn waits on A. They form a loop.
- **Lack of preemption**. Once a resource has been granted to a thread it cannot be taken away.

The Java language does not offer any methods that can prevent a Deadlock. The Deadlocks are a design problem, not a language problem. The design methods that can be used to prevent Deadlock are:

- **Avoidance**. Do not enter a state where a deadlock can occur.
- **Detection and recovery**. If the current state is a deadlock state, than do not go there.
- **Prevention**. Prevent deadlock by removing one of the conditions mentioned above.

Deadlock Example

```
//This is a deadlock example
//It uses two resources A and B
//and two competing threads (racing threads)
class A {

 synchronized void afoo (B b) {
   String tName = Thread.currentThread().getName();
   System.out.println( tName + " is in afoo.");
   try{
     Thread.sleep(5000);
   }catch (Exception e){}
   System.out.println(tName +
                " is trying to call B.lastWork().");
   b.lastWork();
 }

 synchronized void lastWork(){
   System.out.println("A.lastWork() called.");
 }
}

class B {

 synchronized void bfoo (A a) {
   String tName = Thread.currentThread().getName();
   System.out.println( tName + " is in bfoo.");
   try{
     Thread.sleep(7000);
   }catch (Exception e){}
   System.out.println(tName +
                  " is trying to call A.lastWork().");
   a.lastWork();
 }

 synchronized void lastWork(){
   System.out.println("B.lastWork() called.");
 }
}
```

```
public class Deadlock implements Runnable{
 A a = new A();
 B b = new B();

 public Deadlock(){
   Thread.currentThread().setName("MainRunner");
   Thread rt = new Thread(this,"RacingRunner");
   rt.start();
   //get the lock on A in the MainRunner
   a.afoo(b);
   System.out.println("Back in the MainRunner");
 }
 public void run(){
   //get the lock on B in the RacingRunner
   b.bfoo(a);
   System.out.println("Back in the MainRunner");
 }
 public static void main (String [] args){
 //Press Ctrl-C (Ctrl-Break) to break the deadlock
   new Deadlock();
 }
}
/* Output
MainRunner is in afoo.
RacingRunner is in bfoo.
MainRunner is trying to call B.lastWork().
RacingRunner is trying to call A.lastWork().
(waits forever) Ctrl-C pressed
*/
```