CMT219 Algorithms, Data Structures and Programming

Week 5

Multithreading

Multithreaded Applications

- Multithreading naturally suitable for some applications
  - A **web browser** is an example of a **multithreaded** application.
  - When the **browser** is used to download a large document, the user can abort the download by clicking the **stop** button on the **browser's toolbar** or selecting a comparable command from a **browser menu**.
  - In the **browser**, there is a **thread** dedicated to handling user inputs such as button clicks and menu selections and there are **other threads** dedicated to the various tasks associated with downloading sub-parts of the document.
- Threading is also essential to get the most performance out of **multiprocessor** (multicore) machines in a single program.

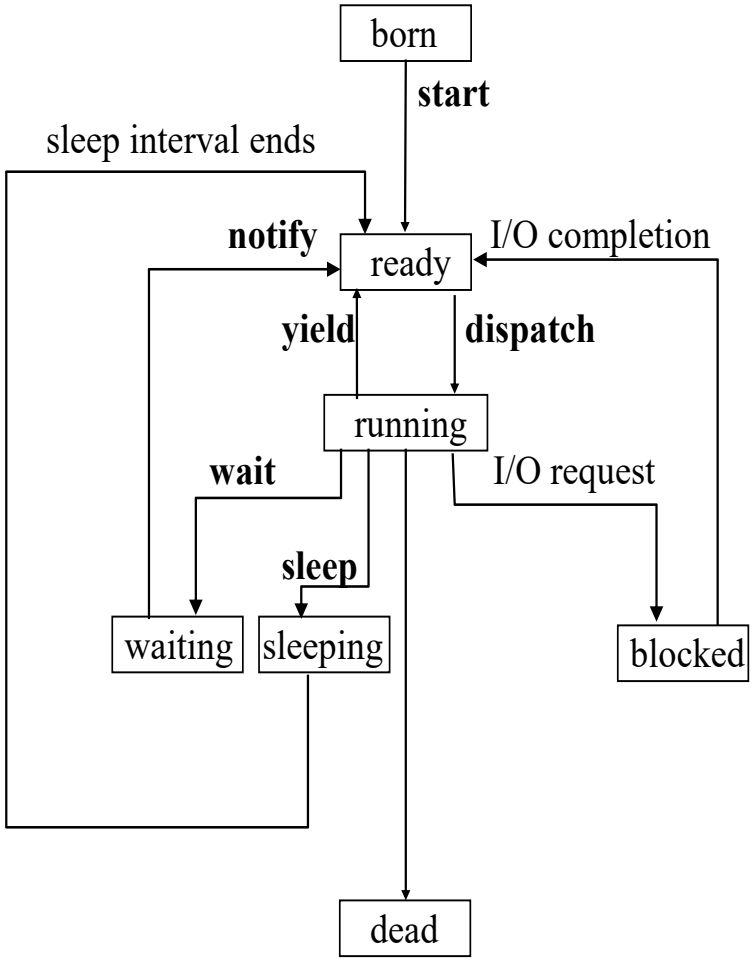
Multithreaded Applications (cont.)

- **Multithreading** allows an application to perform multiple tasks concurrently.
  - An application can have multiple threads $\mathbf{T_1}$, $\mathbf{T_2}$, …, $\mathbf{T_N}$ which handle multiple tasks $\mathbf{Task_1}$, $\mathbf{Task_2}$, …, $\mathbf{Task_N}$, with each thread dedicated to a single task.
  - A single thread might also handle a group of related tasks.
  - Threads allow the programmer to concentrate on a particular task without having to take account of how that task is alternated with other tasks.
  - In order to carry out tasks in parallel, the programmer implements a thread for each of the tasks and starts each thread.

Multithreaded Applications (cont.)

- Sometimes the threads of execution may not be **independent** of one another.
- Some way of **synchronizing** the threads is necessary if
  - one thread needs the result computed by another thread
  - multiple threads simultaneously try to modify a shared object

Life Cycle of a Thread
Thread States



Life Cycle of a Thread

- You define a **run** method which says what the thread is to do.
- When the thread's **start** method is called, the thread enters the ***ready*** state.
  - A ***ready*** thread enters the ***running*** state when the system assigns a processor to the thread (i.e. the thread begins executing)
  - highest priority threads are given a processor in preference
- A thread enters the ***dead*** state when its **run** method completes.

Life Cycle of a Thread (cont.)

- **Blocked** state:
  - A ***running*** thread enters the ***blocked*** state when the thread issues an input/output request.
  - A ***blocked*** thread becomes ***ready*** when the I/O it is waiting for completes.
  - A ***blocked*** thread cannot use a processor even if one is available.
- **Sleeping** state:
  - A ***running*** thread enters the ***sleeping*** state when the thread's **sleep** method is called.
  - A ***sleeping*** thread becomes ***ready*** after the designated sleep time expires.
  - A ***sleeping*** thread cannot use a processor even if one is available.

Life Cycle of a Thread (cont.)

- **Waiting** state:
  - A ***running*** thread enters the ***waiting*** state when the thread calls **wait**, where it waits in a queue associated with a particular object on which the **wait** was called.
  - The first thread in the ***wait*** queue for a particular object becomes ***ready*** on a call to **notify** issued by another thread associated with that object.
  - Waiting state is useful when one thread needs to wait for the data from the other thread to be ready, for instance.

Programs with Multiple Threads

- Execution of single-threaded applications
  - For programs with a **single thread** of execution, the **Java Virtual Machine** (**JVM**) runs the program by executing a single sequence of instructions.
  - In applications the thread begins execution at the first statement in **main**.
  - Termination of a single-threaded **command-line application** occurs when the end of main is reached.

Programs with Multiple Threads (cont.)

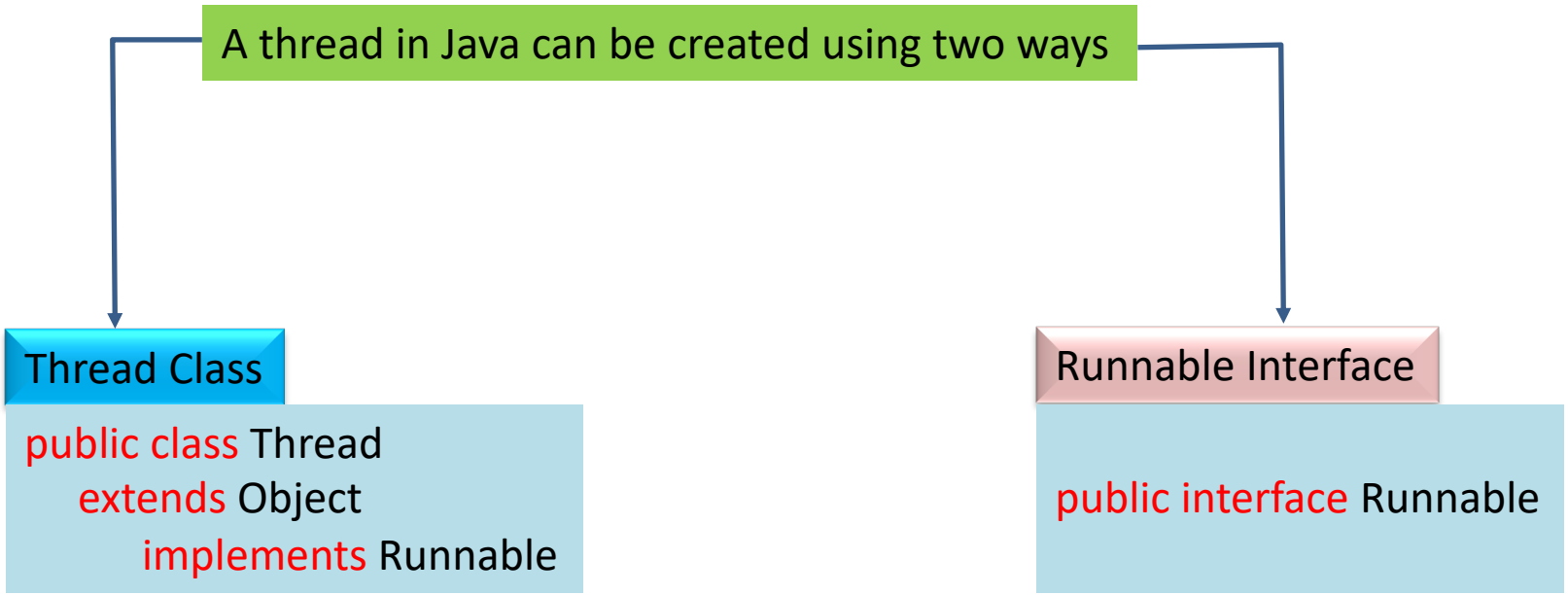
```
public class SingleThread
{
   public static void main( String[] args )
   {
      for ( int i = 0; i < 3; i++ )
         System.out.println( "Hello world" );
   }
   // main exits, thread stops, application halts
}
```

Note: GUI applications also have a separate automatically created event-despatch thread which sits waiting for events and calling appropriate **Listeners**. Such applications are only terminated when **System.exit();** is called.

Programs with Multiple Threads (cont.)

- A command-line program
  - begins as **single-threaded**
  - becomes **multithreaded** if this **single thread** constructs and starts a second **Thread** of execution
- **Thread** is a class in the **java.lang** package.
  - The first thread can construct and start many threads, each of which can construct and start further threads.
  - A started thread executes the body of its **run** method.
  - A second thread of execution may be created by either creating a **subclass** which **extends** the **Thread** class or by creating a **class** that implements the **Runnable** interface.

Creating a Thread



Thread Class

1. Create a thread class
2. Override run() method
3. Create object of the class
4. Invoke start() method to execute the custom threads run()

```
public class MyThread extends Thread {

    public void run()
    {
        System.out.println("My thread");
    }

    public static void main(String[] args)
    {
        MyThread obj = new MyThread();
        obj.start();
    }
}
```

Runnable Interface

1. Create a thread class implementing Runnable interface
2. Override run() method
3. Create object of the class
4. Invoke start() method using the object

```
public class MyThread implements Runnable {

   public void run()
   {
      System.out.println("My thread");
   }

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

Creation of Multithreaded Application

- Approach 1
  - Create a class that **extends** the **Thread** class.
  - Place the code for the task into the **run** method of the subclass so that it overrides the inherited **run** method.
  - Create an object of your subclass.
  - Call the **start** method to start the thread and execute the body of the **run** method.

Creation of Multithreaded Application

- Approach 2
  - Create a class that **implements** the **Runnable** interface
  - Place the code for the task into the **run** method of the class.
  - Create an object of this class.
  - Create a thread object using the previously created Runnable object.
  - Call the **start** method to start the thread and execute the body of the **run** method.
- **Runnable**s are useful when the thread needs to **extend** some other class.

Thread Constructors

- **Thread( )**
- **Thread(Runnable target)**
- **Thread(Runnable target, String name)**
- **Thread(String name)**
- **Thread(ThreadGroup group, Runnable target)**
- **Thread(ThreadGroup group, Runnable target, String name)**
- **Thread(ThreadGroup group, String name)**

- Although all the constructors allocate a new **Thread** object, some have **target** as their **run** object, some have **name** as their Thread name and some belong to the **group** ThreadGroup.

Example (using **Thread** class): **GreetingThread.java**

```
import java.util.Date;
public class GreetingThread extends Thread
{
   // GreetingThread constructor
   public GreetingThread( String aGreeting )
   {
      greeting = aGreeting;
   }

   // this method is executed when the Thread is started
   public void run()
   {
      try
      {
         for ( int i = 0; i <= REPETITIONS; i++ )
         {
            Date now = new Date();           // get current date and time
            System.out.println( now + " " + greeting );  // Output date & time and greeting
            Thread.sleep( DELAY );           // sleep for 1 second
         }
      }
      // exception generated if sleeping thread is interrupted
      catch ( InterruptedException ie )
      {     System.err.println( ie.toString() );      }
   }

   // instance variable
   private String greeting;
   // constants
   private static final int REPETITIONS = 10;
   private static final int DELAY = 1000;
}
```

Example (using **Thread** class): **GreetingThreadTest.java**

```
public class GreetingThreadTest
{
   // main method
   public static void main( String[] args )
   {
      // create first thread
      GreetingThread thread1 = new GreetingThread( "Hello World!" );
      // create second thread
      GreetingThread thread2 = new GreetingThread( "Goodbye World!" );

      thread1.start();    // start first thread
      thread2.start();    // start second thread
   }
   // main thread exits
}
```

Example: **GreetingThreadTest** (cont.)

- Note
  - Although the **main thread** exits, the two **Threads** started in **main** continue to execute.
  - The method **main** in class **GreetingThreadTest** instantiates two **GreetingThread** objects and then starts these threads with the **start** method.

Example (using Runnable interface): Greeting.java

```
import java.util.Date;
public class Greeting implements Runnable
{
   // Greeting constructor
   public Greeting( String aGreeting )
   {  greeting = aGreeting;   }

   // this method is executed when the Thread is started
   public void run()
   {
      try
      {
         for ( int i = 0; i <= REPETITIONS; i++ )
         {
            Date now = new Date();   // get current date and time
            // output date and time and greeting
            System.out.println( now + " " + greeting );
            Thread.sleep( DELAY );   // sleep for 1 second
         }
      }
      // exception generated if sleeping thread is interrupted
      catch ( InterruptedException ie )
      {  System.err.println( ie.toString() );      }
   }

   // instance variable
   private String greeting;
   // constants
   private static final int REPETITIONS = 10;
   private static final int DELAY = 1000;
}
```

Example (using Runnable interface): GreetingTest.java

```
public class GreetingTest
{
   // main method
   public static void main( String[] args )
   {
      // create first greeting object
      Greeting g1 = new Greeting( "Hello World!" );
      // create second greeting object
      Greeting g2 = new Greeting( "Goodbye World!" );

      // create first thread using first greeting object as target
      Thread thread1 = new Thread( g1 );
      // create second thread using second greeting object as target
      Thread thread2 = new Thread( g2 );

      // start first thread
      thread1.start();
      // start second thread
      thread2.start();
   }
   // main thread exits
}
```

Example: **GreetingTest** (cont.)

- Note:
  - Although the **main thread** exits, the two **Threads** started in **main** continue to execute.
  - The method **main** in class **GreetingTest** instantiates two **Greeting** objects and then uses these objects as the **Runnable** targets in the construction of the two **Threads**.
  - The **main thread** executes **main** and whatever other methods are invoked from **main**. A **started** thread executes the **run** method and whatever other methods are invoked from **run**.

Daemon Threads

- The previous applications using **threads** have all used **user threads**.
- An application continues to run as long as at least one of its **user threads** is alive.
- When a **user thread** stops in an application, the **JVM** checks whether any other user threads are still alive.

- A **daemon thread** is one that serves **user threads**.
- If no user threads are still alive, even if daemon threads are running, the application terminates because the **JVM** itself halts.
- Use the **setDaemon(boolean)** method to change the Thread daemon properties before the thread starts.

Example: GreetingThreadTest.java

```
public class GreetingThreadTest
{
   public static void main( String[] args )
   {
      // create first thread
      GreetingThread thread1 = new GreetingThread( "Hello World!" );
      // create second thread
      GreetingThread thread2 = new GreetingThread( "Goodbye World!" );

      thread1.setDaemon( true );      // make first thread a daemon thread
      thread2.setDaemon( true );      // make second thread a daemon thread

      thread1.start();       // start first thread
      thread2.start();       // start second thread

      try
      {
         // user thread main sleeps for 5 seconds
         System.out.println( "Main thread sleeps" );
         Thread.sleep( 5000 );
      }
      catch ( InterruptedException ie )
      {  System.err.println( ie.toString() );      }

      System.out.println( "Main thread wakes up" );
   }
   // the main thread exits and, as it is the only user thread,
   // the daemon threads stop immediately and the application halts
}
```

Daemon Threads (cont.)

- Note:
  - The program sleeps the **main** thread, which is a **user thread**, for 5 seconds
  - The **daemon threads**, **thread1** and **thread2,** continue to run.
  - Because the **main** thread is the application's only **user thread**, the application terminates when **main** returns, even though the daemon threads have not finished.
- A **user thread** constructs other **user threads**, whereas a **daemon thread** constructs other **daemon threads**.
- A constructed thread's **setDaemon** method can be invoked to change a **user thread** to a **daemon thread** (or vice versa) before the thread is started.
- The **JVM**'s garbage collector runs as a **daemon thread** because the **JVM** itself should terminate if no **user threads** are alive.

Thread Priorities

- Thread priorities
  - The **JVM** implements **preemptive**, **priority-based scheduling** for threads.
  - Every **Thread** has an integer priority in the range **MIN_PRIORITY** to **MAX_PRIORITY**.
  - A thread is constructed with the default priority of **NORM_PRIORITY**.
  - Typical values for these are **1**, **5** and **10** respectively.
  - Each new **thread** inherits the **priority** of the **thread** that creates it. A thread's priority can be adjusted with the **setPriority** method.

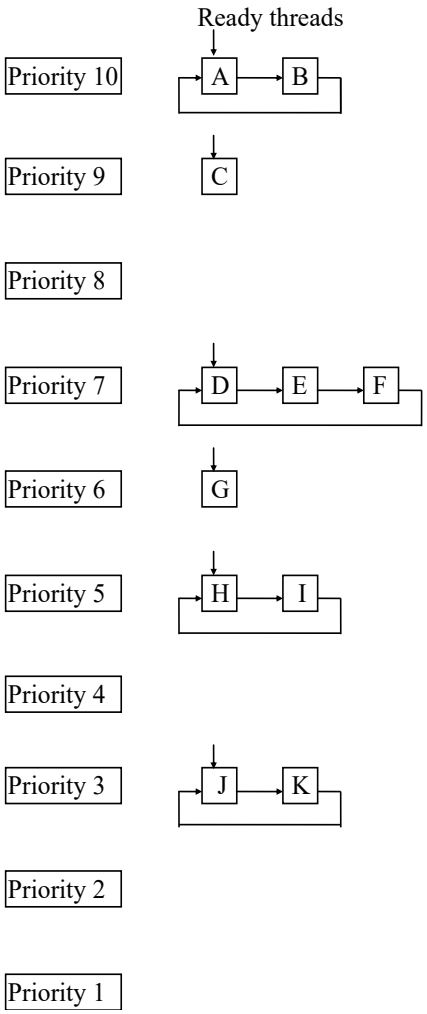
Thread Priorities (cont.)

- Consider a single processor system.
  - When a thread **T** enters a runnable state, e.g. by being started, the **JVM** checks whether the currently executing thread **C** has a lower priority than the thread **T**.
  - If so, the **JVM** preempts the thread **C** in favour of the thread **T**.
  - When a **Java platform** (unusually) does not support **timeslicing**, each **thread** of a set of **equal priority** threads runs to completion before that **thread's peers** get a chance to execute.
  - When **timeslicing** is supported, each **thread** receives a burst of processor time called a **quantum** during which the **thread** can execute.
  - At the completion of the **quantum**, even if that **thread** has not finished executing, the processor is taken away from that **thread** and given to some other **thread** of **equal priority** if one is available.

Thread Priorities (cont.)

- The **Java scheduler** runs the **highest priority thread**.
- If timeslicing is available, the **Java scheduler** ensures that all **threads** of the **highest priority** should each execute for a quantum in a **round-robin** fashion until all **threads** of the **highest priority** have completed execution.
- A **thread** can call the **yield** method to give other **threads** of the **same priority** a chance to run.
- The **yield** method is required on a platform which does not support **timeslicing** in which a **thread** would run to completion before another **thread** of **equal priority** would have an opportunity to run.

Thread Scheduling



Example: PriorityThread.java

```
public class PriorityThread extends Thread
{
   public PriorityThread( int priority, String name )
   {
      setPriority( priority );
      setName( name );
   }

   // this method is executed when the thread is started
   public void run()
   {
      System.out.println(
                  Thread.currentThread().getName() + " Starting");
      // do something which takes a long time
      int j = 100;
      for(long i=0;i<1000000000;i++)
      {
        j = j * 2;
        j = j + 2;
        j = j / 2;
        j = j - 1;
      }
      System.out.println(
                  Thread.currentThread().getName() + " Finished");
   }
}
```

Example: PriorityTest.java

```
public class PriorityTest
{
   public static void main( String[ ] args )
   {
      // start many low priority threads
     for ( int i = 0; i <= NOOFTHREADS; i++ )
     {
        PriorityThread t = new PriorityThread(
            Thread.MIN_PRIORITY, "Low Priority Thread " + i);
        t.start();
     }

      // start a thread with maximum priority
      PriorityThread t2 = new PriorityThread(
            Thread.MAX_PRIORITY, "High Priority Thread **" );
      t2.start();
   }

   // main thread exits
   private static final int NOOFTHREADS = 50;
}
```

Terminating Threads Early

- A thread terminates when its **run** method returns.
  - Sometimes we wish to **stop** a thread we have started, before it finishes normally in this way.
  - For example, we may start a thread to download a Web page, and the user may wish to cancel before the page has arrived.
- The basic way to safely stop a thread is for the thread to periodically (for example, once round some loop of work) **check a variable** inside the thread which can be set by some other thread, and indicates whether a stop has been requested.
- If the thread is **not runnable**, e.g. it is sleeping, however, we also need to force it to wake up so we can stop it. This can be done using the **interrupt** method.

Example: GreetingThread.java

```
import java.util.Date;
public class GreetingThread extends Thread
{
   // GreetingThread constructor
   public GreetingThread( String aGreeting ) {
      greeting = aGreeting;
   }
   // this method is executed when the thread is started
   public void run()
   {
      stopRequested = false;
      try {
         for ( int i = 0; i <= REPETITIONS && ! stopRequested; i++ ) {
            Date now = new Date();           // get current date and time

            System.out.println( now + " " + greeting );  // output date & time and greeting
            Thread.sleep( DELAY );           // sleep for 1 second
         }
      }
      // exception if sleeping thread is interrupted
      catch ( InterruptedException ie ) {
        System.out.println("Interrupted in sleep");
      }
      // clean up procedure
   }
   public void requestStop()
   {
     stopRequested = true;
     interrupt();
   }
   // instance variable
   private String greeting;
   private Boolean stopRequested;
   // constants
   private static final int REPETITIONS = 10;
   private static final int DELAY = 1500;
}
```

Example: GreetingThreadTest.java

```
public class GreetingThreadTest
{
   public static void main( String[] args )
   {
      // create 1st greeting thread
      GreetingThread thread1 =
                  new GreetingThread( "Hello World!" );
      // create 2nd greeting thread
      GreetingThread thread2 =
                  new GreetingThread( "Goodbye World!" );

      thread1.start();       // start 1st thread
      thread2.start();       // start 2nd thread

      try
      {
         System.out.println( "Main thread sleeps" );
         Thread.sleep( 5000 );        // main thread sleeps for 5 seconds
      }
      catch ( InterruptedException ie )
      {  System.err.println( ie.toString() );      }

      System.out.println( "Main threads wakes up" );
      // requestStop is sent to first thread
      thread1.requestStop();
   }
   // main thread exits
}
```

Example: GreetingThreadTest

- Results (example)

```
Main thread sleeps
Wed Apr 17 17:21:58 BST 2013 Hello World!
Wed Apr 17 17:21:58 BST 2013 Goodbye World!
Wed Apr 17 17:22:00 BST 2013 Goodbye World!
Wed Apr 17 17:22:00 BST 2013 Hello World!
Wed Apr 17 17:22:01 BST 2013 Hello World!
Wed Apr 17 17:22:01 BST 2013 Goodbye World!
Wed Apr 17 17:22:03 BST 2013 Goodbye World!
Wed Apr 17 17:22:03 BST 2013 Hello World!
Main threads wakes up
Interrupted in sleep
Wed Apr 17 17:22:04 BST 2013 Goodbye World!
Wed Apr 17 17:22:06 BST 2013 Goodbye World!
Wed Apr 17 17:22:07 BST 2013 Goodbye World!
Wed Apr 17 17:22:09 BST 2013 Goodbye World!
Wed Apr 17 17:22:10 BST 2013 Goodbye World!
Wed Apr 17 17:22:12 BST 2013 Goodbye World!
Wed Apr 17 17:22:13 BST 2013 Goodbye World!
```

Joins

- Sometimes one thread needs to **wait** for various **other threads** to **complete** their work before it can continue: typically, they are producing some data it needs.
  - For example, a web browser may use several **downloading threads** to download individual images and pieces of text for a web page.
  - The page can only be drawn by a **rendering thread** when **all** of the downloading threads have **finished**, and all page elements are available (it needs to know how big each image is, etc, before it can lay them out).
  - Real browsers are smarter than this, and figure out image sizes before getting the whole image. Let's ignore this for now!
- The **join();** method lets one thread **wait** for **another thread** to **complete** before it continues.
- If a given thread must wait for **several** threads to complete, it should **join** them **all**.

Example: Slave.java

```
public class Slave extends Thread
{
   // Slave constructor
   public Slave( String name, int numberOfTimes )
   {
      super( name );
      repeats = numberOfTimes;
   }

   public void run()
   {
      try
      {
         for ( int i = 1; i <= repeats; i++ )
         {
            // output thread name and thread group
            System.out.println( getName() + "." + i);
            Thread.sleep( DELAY );          // sleep for 1 second
         }
      }
      // exception generated if sleeping thread is interrupted
      catch ( InterruptedException ie )
      {  System.out.println("Interrupted while sleeping" );      }
   }

   // instance variables
   private int repeats;
   // constants
   private static final int DELAY = 1000;
}
```

Example: Master.java

```
public class Master
{
   public static void main( String[ ] args )
   {

      Slave thread1 = new Slave ( "Thread 1", 5 );      // create 1st thread
      Slave thread2 = new Slave ( "Thread 2", 2 );      // create 2nd thread
      Slave thread3 = new Slave ( "Thread 3", 7 );      // create 3rd thread

      thread1.start();            // start 1st thread
      thread2.start();            // start 2nd thread
      thread3.start();            // start 3rd thread

      // In principle, we may be interrupted while waiting for in a join,
      // so we need a try-catch block.      Actually, we know nothing can
      // interrupt main thread. so we just use an empty catch block.
      try
      {
        // wait for each of the other threads to finish
        thread1.join();
        thread2.join();
        thread3.join();
      }
      catch (InterruptedException e)
      {
      }

      // do something that needs all other threads to have finished
      System.out.println( "All work finished" );
   }
   // main thread exits
}
```

Joins

- Example results:

```
Thread 1.1
Thread 2.1
Thread 3.1
Thread 1.2
Thread 2.2
Thread 3.2
Thread 1.3
Thread 3.3
Thread 3.4
Thread 1.4
Thread 1.5
Thread 3.5
Thread 3.6
Thread 3.7
All work finished
```

Thread Synchronization

- **Threads** execute **independently** in the JVM, although there are ways to coordinate or **synchronize** thread execution.
  - Each thread has its **own copy** of **local variables** in whatever methods the thread happens to execute.
  - A thread cannot access the **local** variables of another thread
  - The local variables are said to be **thread safe** as the programmer is not required to provide any special thread synchronization to prevent one thread from accessing another thread's local variables.
- If two threads share a **common object**, synchronization is necessary

Thread Synchronization (cont.)

- Problems may occur, for example, if
  - **one thread** tries **get information** from an object, while **another thread** is part way through **updating** it
  - **two threads** try to **update** the same object at the same time, leaving it in an **inconsistent** state
- The solution is to make sure that during **critical** pieces of code, only one thread can access the object. **Locks** are used to do this. Java provides built-in locks through the **synchronized** keyword

Example: BankAccount.java

```
public class BankAccount
{
   // default constructor
   public BankAccount()
   {  balance = 0;   }

   // method for depositing money
   public void deposit( double amount )
   {
      System.out.print( "Depositing " + amount );
      double newBalance = balance + amount;
      System.out.println( ", new balance is " + newBalance );
      balance = newBalance;
   }

   // method for withdrawing money
   public void withdraw( double amount )
   {
      System.out.print( "Withdrawing " + amount );
      double newBalance = balance - amount;
      System.out.println( ", new balance is " + newBalance );
      balance = newBalance;
   }

   // method for getting a balance
   public double getBalance()
   {  return balance;   }

   // instance variable
   private double balance;
}
```

Example: DepositThread.java

```
public class DepositThread extends Thread
{
   // DepositThread constructor
   public DepositThread( BankAccount anAccount, double anAmount )
   {
      account = anAccount;
      amount = anAmount;
   }

   // this method is executed when the thread is started
   public void run()
   {
      try
      {
         for ( int i = 0; i <= REPETITIONS && ! isInterrupted(); i++ )
         {
            account.deposit( amount );       // deposit money into account
            Thread.sleep( DELAY );           // sleep for 5 milliseconds
         }
      }
      // exception generated if sleeping thread is interrupted
      catch ( InterruptedException ie )
      {  System.out.println("Interrupted while sleeping" );     }
   }

   // instance variables
   private BankAccount account;
   private double amount;
   // constants
   private static final int REPETITIONS = 500;
   private static final int DELAY = 5;
}
```

Example: WithdrawThread.java

```
public class WithdrawThread extends Thread
{
   // WithdrawThread constructor
   public WithdrawThread( BankAccount anAccount, double anAmount )
   {
      account = anAccount;
      amount = anAmount;
   }

   // this method is executed when the thread is started
   public void run()
   {
      try
      {
         for ( int i = 0; i <= REPETITIONS && ! isInterrupted(); i++ )
         {
            account.withdraw( amount );      // withdraw money from account
            Thread.sleep( DELAY );           // sleep for 5 milliseconds
         }
      }
      // exception generated if sleeping thread is interrupted
      catch ( InterruptedException ie )
      {  System.out.println("Interrupted while sleeping" );      }
   }

   // instance variables
   private BankAccount account;
   private double amount;
   // constants
   private static final int REPETITIONS = 500;
   private static final int DELAY = 5;
}
```

Example: BankAccountThreadTest.java

```
public class BankAccountThreadTest
{
   public static void main( String[] args )
   {
      // create a new bank account
      BankAccount account = new BankAccount();
      // create a deposit thread
      DepositThread deposit = new DepositThread( account, 100 );
      // create a withdraw thread
      WithdrawThread withdraw = new WithdrawThread( account, 100 );

      // start deposit thread
      deposit.start();
      // start withdraw thread
      withdraw.start();
   }
}
```

Example: BankAccountThreadTest

- When this application is run, the **deposit** and **withdraw** methods of the **BankAccount** class are called from the threads **DepositThread** and **WithdrawThread** respectively.
- Overall, the DepositThread in principle **adds** to the account exactly **as much as** the WithdrawThread **removes**. At the end, the balance should still be 0.

- In practice this is (probably) **not** what we see. Why?

Example: BankAccountThreadTest (cont.)

- Suppose a withdraw is done by the **WithdrawThread** part way through a deposit done by the **DepositThread** (or vice versa):
  - In the deposit method we execute the line
  - **double newBalance = balance + amount;**
  - Suppose the **WithdrawThread** executes **withdraw**. This causes the **balance to change**.
  - However, we subsequently execute in the **deposit** method
  - **balance = newBalance;**
  - which **overwrites** the change made by the withdraw thread and loses its update.

Synchronised Example: BankAccount.java

```
public class BankAccount
{
   // default constructor
   public BankAccount()
   {  balance = 0;   }

   // method for depositing money
   public synchronized void deposit( double amount )
   {
      System.out.print( "Depositing " + amount );
      double newBalance = balance + amount;
      System.out.println( ", new balance is " + newBalance );
      balance = newBalance;
   }

   // method for withdrawing money
   public synchronized void withdraw( double amount )
   {
      System.out.print( "Withdrawing " + amount );
      double newBalance = balance - amount;
      System.out.println( ", new balance is " + newBalance );
      balance = newBalance;
   }

   // method for getting a balance
   public double getBalance()
   {  return balance;          }

   // instance variable
   private double balance;
}
```

Synchronised Example: BankAccount

- Each **object** has a **lock** and, by default, an **object** is **unlocked**.
- A **thread** obtains the **lock on** an **object** by executing a **synchronized** method.
- The **thread** that is executing the **synchronized** method owns the **lock** until it returns from the method and thereby **unlocks** the **object**.
- When an **object** is **locked** by one thread, no other thread can enter a **synchronized** method for that **object**.
- If another thread makes a call to a **synchronized** method for that **object**, this other thread is automatically made to wait until the first thread has **unlocked** the **object** again.

Deadlocks

- Object **locks** and **synchronized** methods can be used to ensure that **objects** are in a consistent state when several threads access them.
- However, suppose one thread has **locked** an **object** and is waiting for another thread to do some essential work.
- If the other thread is currently waiting to use, and hence to **lock** the same **object**, then neither of the two threads can proceed and the program is locked up.
- Such a situation is called a **deadlock** or **deadly embrace**. To the user, the program **hangs forever**. It is stuck.
- **Avoiding** deadlocks is tricky in general.

Example: BankAccount.java

```
public class BankAccount
{
   // constructor
   public BankAccount(int initialAmount)
   {  balance = initialAmount;   }

   // method for depositing money
   public synchronized void deposit( double amount )
   {
      System.out.print( "Depositing " + amount );
      double newBalance = balance + amount;
      System.out.println( ", new balance is " + newBalance );
      balance = newBalance;
   }

   // method for withdrawing money
   public synchronized void withdraw( double amount )
   {
      System.out.print( "Withdrawing " + amount );
      double newBalance = balance - amount;
      System.out.println( ", new balance is " + newBalance );
      balance = newBalance;
   }

   // method for transferring money to another account
   public synchronized void transfer( BankAccount to, double amount )
   {
      System.out.println( "Transferring " + amount );
      this.withdraw(amount);
      synchronized(to)
      { to.deposit(amount           );       }
   }

   // method for getting a balance
   public double getBalance()
   {  return balance;   }

   // instance variable
   private double balance;
}
```

Example: MoverThread.java

```
public class MoverThread extends Thread
{
   // MoverThread constructor
   public MoverThread( BankAccount fromAccount, BankAccount toAccount,
                              double anAmount )
   {
      source = fromAccount;
      destination = toAccount;
      amount = anAmount;
   }

   // this method is executed when the thread is started
   public void run()
   {
         for ( int i = 0; i <= REPETITIONS && ! isInterrupted(); i++ )
         {
            // move money from source to destination
            source.transfer( destination, amount );
         }
   }

   // instance variables
   private BankAccount source, destination;
   private double amount;
   // constants
   private static final int REPETITIONS = 500;
   private static final int DELAY = 5;
}
```

Example: BankAccountThreadTest.java

```
public class BankAccountThreadTest
{
   public static void main( String[] args )
   {
      // create two new bank accounts
      BankAccount myAccount = new BankAccount(1000);
      BankAccount yourAccount = new BankAccount(1000);

      // create and start thread moving money from me to you
      System.out.println("Starting first mover thread");
      MoverThread meToYou =
                new MoverThread( myAccount, yourAccount, 75 );
      meToYou.start();

      // wait a short time, then start second thread
      try
      {
        Thread.sleep(2);
      }
      catch ( InterruptedException ie )
      {  System.out.println( "Interrupted while sleeping" );      }

      // create and start thread moving money from you to me
      System.out.println("Starting second mover thread");
      MoverThread youToMe = new MoverThread( yourAccount, myAccount, 99 );
      youToMe.start();
   }
}
```

Deadlock

- Two ways of using **synchronized** keyword
  - Put **synchronized** in method definition: entering the method requires the lock of the object
  - Use **synchronized (object) { … }** block: entering the block requires the lock of the specified **object**
- Can you see why the previous code may lead to deadlock?