Chapter

30

SECTION VI: CLIENT SIDE PROGRAMING

Java Threads

A thread is often defined as the **single, sequential flow** of control within any computer program. It is also referred to as a **thread of control**. This is the linear, hierarchic, top down, interpretation and execution of any computer program. This is the way that the CPU ensures that all lines in the computer program are definitely executed.

Java provides built-in support for programmers who need to create **multithreaded** programs. **A multithreaded program contains two [or more] individual, internal, code blocks that can** run **concurrently**. Each such code block is called a **thread**. Each thread thus defines a separate path of the same program's execution. Thus it appears that program code spec is multitasking.

Even if a programming environment [like Java] has built-in multitasking capabilities. Multitasking must also be supported by the operating system. Virtually all the modern operating systems do so. There are two distinct types of multitasking:

- Process based
- Thread based

Process based multitasking is perhaps more familiar than thread based. A **process** is essentially a program that is executing. Process based multitasking is a feature that allows a computer to run two or more programs concurrently. For example: Process based multitasking enables running a Java compiler at the same time that a text editor is being used. In process based multitasking a program is the **smallest** unit of code that can be dispatched by the scheduler to the CPU.

In a **thread-based**, multitasking environment, the thread is a smallest unit of **dispatchable** code to the CPU. Hence, a single program can perform two or more tasks simultaneously. For example, a text editor can format text while it is printing a document, as long as these two actions are being performed by **two separate** threads originated and owned by the text editor.

Multitasking threads consume much fewer systems resources than multitasking processes. Processes are heavyweight tasks that require their own area of RAM memory. **Interprocess** communication is expensive and limited. Switching between processes **is costly** in terms of CPU **clock cycles** and the consumption of RAM. Threads, on the other hand, are lightweight. They share the same address space and can cooperatively share the same heavyweight processes on demand. **Interthread** communication is inexpensive. Switching from one thread to the next is amazingly low in terms of CPU cycles and RAM consumption when compared to process switching.

There are multiple advantages of multithreading. Multithreaded programs always result in better utilization of system resources including CPU clock cycles because a thread can grab the CPU when the current thread using the CPU is waiting or blocked.

Java was designed from the ground up with multithreading in mind. Hence, Java has strong multithreading support built in. This allows the creation of robust, multithreaded Java applications.

What Is Multithreading?

Due to extremely high speeds of execution of modern CPU's, they can handle multiple flows of control within the execution of a single program. A specific flow of control, within multiple control flows belonging to a single program is a thread. Threads can share many system resources such as memory and file content. Therefore, a thread is called a **lightweight process**. A thread is easier to create and destroy than a process because whole lot less of

resource management is involved.

Program execution may spend a large portion of its time just waiting. For example: waiting for resources to become accessible via an I/O operation [i.e. a hard disk read operation] or waiting for a timeout to occur to begin another job. To improve CPU utilization, all the tasks with potentially long waits can be run as **separate threads**. Once a task starts waiting for something to happen, the JVM can automatically choose another **runnable** task for execution.

The Thread Control Methods

There are several methods of the Thread class that can be used to control the execution of a thread. The commonly used ones are:

- **start():** Used to start thread execution. When start() is called it immediately calls the **run()** method, nothing else
- **stop():** Used to stop thread execution, no matter what the thread is doing. The thread is then considered **dead**. The internal states of the thread are cleared and the resources consumed by the thread freed to be reused
- **suspend():** Used to temporarily stop thread execution. All the resources and states of the thread are retained. This method causes a running thread to pause
- **resume():** Used to re-start a suspended thread. The resumed thread will be scheduled to run. In pre-emptive multitasking if the resumed thread has a higher priority than the one currently running, the thread currently running will be preempted, otherwise, the resumed thread is marked as **just-resumed** and will wait in the thread queue for its turn to be executed by the CPU
- **sleep(long sleep_time_in_milliseconds):** A method of the thread class that instructs the Java runtime to put a thread to sleep for a specified period of time. A **InterruptedException**, may be thrown while the thread sleeps. A **try-catch-finally** code block needs to be defined to handle this exception or the enclosing method needs to have this exception coded in its **throws** clause
- **join():** Using the join() method of the thread class, one thread can be made to wait for another to complete. join() guarantees that the second thread will not begin execution until the thread to which it is joined completes its tasks as defined in its run() method. join() can also take a *milliseconds* argument which will cause the joined thread to wait for the designated time period after the thread to which it is joined completes
- **yield():** A thread class method that temporarily stops a thread's execution and puts it at the end of the thread queue to wait for another turn to execute. It is used to make sure other threads of the same priority have a chance to run

The Thread Life Cycle

Every thread, after its creation and before its destruction, will exist in one of four states:

- Newly created
- Runnable
- Blocked
- Dead

Newly Created Threads

A thread enters the newly created state immediately at the moment it is created. A thread is created when the **new** thread() statement is executed. In this state, the thread's local data members are allocated and initialized.

Only when the **start()** method, calls the **run()** method, is the thread in a **runnable** state.

Runnable Threads

When a thread is flagged as **runnable**, it is ready to execute its code base and can now be in one of two states viz. **running** or **queued**.

When a thread is being executed by the CPU it is in its running state. When a thread is in its queued state, it is waiting in a queue and competing for its turn to be executed by the CPU.

A runtime scheduler controls the transition between these two sub-states. However, a thread can call its **yield()** method to voluntarily move itself to a queued state from within a running state.

Blocked Threads

A thread enters a blocked state when one of the following events occur:

- The thread itself [*or another thread*] calls its **suspend()** method
- The thread calls an object's **wait()** method
- The thread itself calls its **sleep()** method

- The thread is waiting for some I/O operation to complete

A thread in a blocked state is not scheduled for running.

A thread will go back to a **runnable** state, [competing for CPU cycles], when:

- Another thread calls the suspended thread's **resume()** method
- A thread is put to sleep and the specified sleeping time elapses
- The thread is blocked on I/O operation and the specified I/O operation completes

Dead Threads

A tread enters a dead state when it finishes its execution or is stopped by another thread calling its **stop()** method.

REMINDER

To find out whether a thread is alive, isAlive() can be used and passed the thread handle. isAlive() will return **true** if the thread is alive otherwise it will return **false**.

The Main Thread

When any Java program commences execution, one thread begins running immediately. This is the **main thread** of the Java program. This thread starts automatically and immediately when the program begins its execution. The main thread is important for the following reasons:

- It is the thread from which all other **child** threads are spawned
- It will be the last thread to finish execution. When the main thread **stops**, the program **terminates**

Although the main thread is created automatically when a Java program begins execution, it can be controlled through the **Thread** object. To do this, obtain a reference to the main thread by calling **currentThread()**, which is a **public static** member of **Thread**. Its general form is shown here:

```
static Thread currentThread()
```

currentThread() returns a reference to the current thread. If this is the **first line in the Java program** a reference to the **main** thread is returned. Once a reference to the main thread is gained, this thread can be controlled just like any other thread.

Example:
The following example controls the main thread of a Java program.

Code spec for MainThread.java:

```
class MainThread
{
   public static void main(String args[])
   {
        Thread threadMain = Thread.currentThread();
        System.out.println("Current thread: " + threadMain);
        try
        {
            for(int n = 5; n > 0; n--)
            {
                System.out.println(n);
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException interrupExcep)
        {
            System.out.println ("The main thread interrupted");
        }
        threadMain.setName("Second Thread");
        System.out.println("New name for the thread is: " + threadMain);
   }
}
```

Explanation:
In the above code spec:

```
Thread threadMain = Thread.currentThread();
```

An object of Thread is created. A reference to the current thread [the main thread, in this case] is obtained by calling **currentThread()** and this reference is stored in the local variable **threadMain**.

```
try
{
   for(int n = 5; n > 0; n--)
   {
        System.out.println(n);
        Thread.sleep(1000);
   }
}
catch (InterruptedException interrupExcep)
{
   System.out.println ("The main thread interrupted");
```

```
}
threadMain.setName("Second Thread");
```

Next, the application displays information about the thread. The application then calls the **setName()** method to change the internal name of the thread. Information about the thread is then redisplayed. Next, a For loop counts down from five, pausing one second between each line. The pause is accomplished by the **sleep()** method. The argument to **sleep()** specifies the delay period in milliseconds.

The **sleep()** method in **Thread** might throw an **InterruptedException**. This would happen if some other thread wanted to interrupt this sleeping one. This example just prints a message if the sleeping thread gets interrupted. In a real life program, this would be handled differently.

Once the .java file is ready, it needs to be compiled by the Java compiler [javac] and then the .class file is interpreted by the Java interpreter [java]. After compiling and executing the window as shown in diagram 30.1 appears.

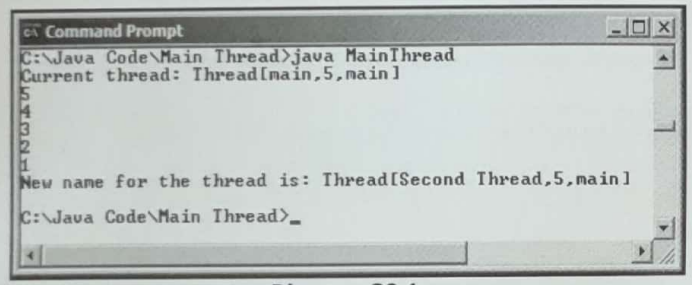

Diagram 30.1

Notice the output produced when **n** is used as an argument to **println()**. This displays in order: the name of the thread, its priority and the name of its group. By default, the name of the main thread is **main**. Its priority is **5**, which is the default value. And **main** is the name of the group to which this thread belongs.

A **thread group** is a data structure that controls the state of a collection of threads as a whole. After the name of the thread is changed, the new name of the thread is displayed.

Let's look more closely at the methods defined by **Thread** that are used in the program. The **sleep()** method causes the thread from which it is called to suspend execution for the specified period of **milliseconds**. Its general form is shown here:

```
static void sleep(long milliseconds) throws InterrruptedException
```


The time that a thread is suspended is specified in **milliseconds**. This method may throw an **InterruptedException**.

The **sleep()** method has a second form, shown next, which allows specifying the period in terms of milliseconds and nanoseconds:

```
static void sleep(long milliseconds, int nanosenconds) throws InterruptedException
```

REMINDER

This second form is useful only in environments that allow timing periods as short as nanoseconds.

As the preceding program shows, the name of a thread can be set by using **setName()**. The name of a thread can be obtained by calling the **getName()** method. These methods are members of the **Thread** class and are declared as follows:

```
final void setName(String threadName)
```

Here, **threadName** specifies the name of the thread:

```
final String getName()
```

Creating A Thread

To create a thread instantiate an object of the type **Thread**. These are the ways that Java defines to accomplish this:

- Implement the **Runnable** interface
- Extend the **Thread** class

The **Thread** class is contained within the **java.lang** package.

REMINDER

The classes contained within **java.lang** are always accessible in any **.class** file irrespective whether java.lang was explicitly imported while creating its .java program.

Implementing Runnable

A simple way to spawn a thread is to create a class that implements the **Runnable** interface. **Runnable** abstracts a unit of executable code. To implement **Runnable** the class needs to

implement a single method called **run()**, as follows:

```
public void run()
```

run(), will contain code spec that defines the functionality of the new thread. Remember **run()** can call other methods, access other classes and declare variables, exactly like the main **run()** can call other methods, access other classes and declare variables, exactly like the main thread can. **run()** also establishes the start of another, concurrent thread, of execution within the program. This thread will end when the **run()** code spec execution completes.

After defining a class that implements **Runnable**, instantiate an object of the type **Thread** from within that class. **Thread** defines several constructors.

```
Thread(Runnable threadOb, String threadName)
```

In this Thread constructor, **threadOb** is an instance of a class that implements the **Runnable** interface. **threadOb** defines where execution of this thread will begin. Replace the string **threadName** with the name of the new thread.

Once spawned the new thread will not start running until its **start()** method is called, which is declared within **Thread**. **start()** simply executes a call to **run()**.

The **start()** method is shown below:

```
void start();
```

Example:
The following example creates a clock by implementing Runnable.

Code spec for Clock.java:

```
import java.awt.*;
import java.util.*;
import java.text.DateFormat;

public class Clock extends java.applet.Applet implements Runnable
{
   private Thread threadClock = null;

   public void init()
   {
      setBackground(Color.white);
   }

   public void start()
   {
      if (threadClock == null)
      {
         threadClock = new Thread(this, "Clock");
         threadClock.start();
      }
   }
   public void run()
   {
      Thread threadCurrent = Thread.currentThread();
      while (threadClock == threadCurrent)
      {
         repaint();
         try
         {
            Thread.sleep(1000);
         }
         catch (InterruptedException e)
         {
         }
      }
   }

   public void paint(Graphics g)
   {
      Font font = new Font("Comic Sans MS", Font.BOLD, 28);
      g.setFont(font);
      g.setColor(Color.blue);
      Calendar calci = Calendar.getInstance();
      Date currentTime = calci.getTime();
      DateFormat dtFormat = DateFormat.getTimeInstance();
      g.drawString(dtFormat.format(currentTime), 95, 100);
   }

   public void stop()
   {
      threadClock = null;
   }
}
```


Explanation:
In the above code spec:

```
public class Clock extends java.applet.Applet implements Runnable
{
   ...
}
```

The class named Clock is created which extends the java.applet.Applet class and implements Runnable abstracts. Runnable interface only provides a design upon which classes should be implemented.

private Thread threadClock = null;

A private object of Thread named threadClock is created and it holds the empty value.

```
public void init()
{
   setBackground(Color.white);
}
```

The init() method is called exactly once in an applet's life, when the applet is first loaded. Here, the init() method is used to set the background color to white.

```
public void start()
{
   if (threadClock == null)
   {
       threadClock = new Thread(this, "Clock");
       threadClock.start();
   }
}
```

The start() method is called at least once in an applet's life, when the applet is started or restarted. Many applets written does not have explicit start() methods and it merely inherit one from their superclass. A start() method is often used to start any threads the applet needs while it runs.

```
public void run()
{
   Thread threadCurrent = Thread.currentThread();
   while (threadClock == threadCurrent)
   {
       repaint();
       try
       {
           Thread.sleep(1000);
       }
       catch (InterruptedException e)
       {
       }
   }
}
```

When an object implementing interface Runnable is used to create a thread, starting the thread causes the object's run() method to be called in that separately executing thread.

The run() method loops until the browser asks it to stop. During each iteration of the loop,

the clock repaints its display. In the above code spec, the sleep() method needs to throw an InterruptedException in case the current thread is interrupted by another thread and therefore the sleep() method is coded inside the try/catch block.

```
public void paint(Graphics g)
{
   Font font = new Font("Comic Sans MS", Font.BOLD, 28);
   g.setFont(font);
   g.setColor(Color.blue);
   Calendar calci = Calendar.getInstance();
   Date currentTime = calci.getTime();
   DateFormat dtFormat = DateFormat.getTimeInstance();
   g.drawString(dtFormat.format(currentTime), 95, 100);
}
```

The paint() method figures out what time it is, formats it in a localized way and displays it.

```
public void stop()
{
   threadClock = null;
}
```

The stop() method is called at least once in an applet's life, when the browser leaves the page in which the applet is embedded. The applet's start() method is called if at some later point the browser returns to the page containing the applet. In the above code spec, the stop() method is used to pause the running threads. When the applet is stopped, it should not use any CPU cycles.

Lifecycle Of A Thread

A Thread continues to execute until one of the following happen:

- It returns from its target run() method
- It's interrupted by an unhandled exception
- It's stop() method is called

If the run() method of a thread never terminates and the application that started the thread never calls it's stop() method then that thread continues its existence, even after the application that spawned it finishes. Hence, developers creating multi threaded applications must be aware of how each thread manually spawned, eventually terminates so that there are no orphaned threads that unnecessarily consume CPU and other resources.

Often, a developer really needs to create background threads that do simple, periodic tasks in an application. The setDaemon() method can be used to mark a Thread as a daemon thread

that should be killed and discarded when no other application threads remain by Java's automatic garbage clearing system. Normally, the Java interpreter continues to run until all threads have completed. But when daemon threads are the only threads still alive, the interpreter will terminate such threads and exit.

The following is the example of using daemon threads:

```
class Clock extends Thread
{
   Clock()
   {
      setDaemon(true);
      start();
   }

   public void run()
   {
      . . .
   }
}
```

In the above example, the Clock thread is set to a daemon status when it is created. If any Clock threads remain when the application is otherwise complete, the Java runtime kills them. So the developer does not have to worry about cleaning them up.

Daemon threads are primarily useful in standalone Java applications and in the implementation of the Java system itself. Daemon threads are not useful in applets. Since an applet runs inside of another Java application, any daemon threads it creates will continue to live until the controlling application exits [probably not the desired effect].

Code spec of index.jsp:

```
<%@ page language="java" contentType="text/html" import="java.lang.*" %>
<HTML>
   <HEAD>
      <TITLE>Welcome To The World Of Threads</TITLE>
   </HEAD>
   <BODY>
      <jsp:plugin type="applet" code="Clock.class" codebase="/MyWebApplication/"
                  width="300" height="200">
         <jsp:fallback>Unable To load Applet</jsp:fallback>
      </jsp:plugin>
   </BODY>
</HTML>
```

Make a war file and deploy the application. After deploying the file, move the class file to the root directory of the web application as explained earlier.

Next, run the JSP file in the browser as shown in diagram 30.2.

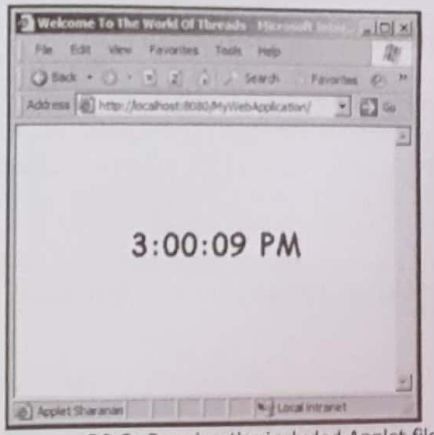

Diagram 30.2: Running the included Applet file

Extending The Thread Class

To create a thread, extend the Thread class, then to create an instance of that class. The class that extends Thread must call start() to begin execution of the new thread. The run() method must be overridden with code spec that defines the functionality of the new Thread. This is the entry point for the new thread.

The following example creates a subclass of the Thread class and overrides the run() method of the Thread class.

The run() method is called when the thread starts executing. To start the thread, call the start() method of the Thread class.

```
class MyExtendedThread extends Thread
{
/* Constructor */
   public MyExtendedThread()
   {
   /* Initializing the code */
   }
/* run() thread */
   public void run()
```

```
    {
    /* The code the thread should execute */
    }
}
```

Once Thread is **subclassed**, An object of this class is instantiated to start the thread, as follows:

```
MyExtendedThread myThread = new MyExtendedThread(this);
myThread.start();
```

- The first statement creates an object of the **MyExtendedThread** class and calls the constructor of the class by passing the reference of the thread object to **MyExtendedThread** class using the **this** reference
- The second statement uses the **start()** method of the **Thread** class, which is overridden in the **MyExtendedThread** class. The **start()** method in turn calls the **run()** method

Example:
The following is an example, which uses **extends Thread**. Two files are created viz. SimpleThread.java, which sets the name of the thread and prints the name of the thread randomly 10 times and CallSimpleThread.java, which starts the SimpleThread file two times.

Code spec for SimpleThread.java:

```
public class SimpleThread extends Thread
{
    public SimpleThread(String strThread)
    {
    /* Setting the name of the thread to strThread */
        super(strThread);
    }

    public void run()
    {
    /* Printing the value of i variable along with the name of the
    thread */
        for (int i = 0; i < 10; i++)
        {
            System.out.println(getName() + ": " + i);
            try
            {
                sleep((long)(Math.random() * 1000));
            }
            catch (InterruptedException e)
            {
            }
    }
```



```
        System.out.println(getName() + ": 10");
    }
}
```

Explanation:
In the above code spec:

```
public SimpleThread(String strThread)
{
    super(strThread);
}
```

A constructor [a new instance] of the same class is created which accepts the name of the thread. After accepting the name of the thread it sets the same using the super() method i.e. the super() method initializes the thread.

```
public void run()
{
    for (int i = 0; i < 10; i++)
    {
        System.out.println(getName() + ": " + i);
        try
        {
            sleep((long)(Math.random() * 1000));
        }
        catch (InterruptedException e)
        {
        }
    }
    System.out.println(getName() + ": 10");
}
```

The run() method loops until the browser asks it to stop. During each iteration of the loop, the name of the thread is displayed randomly.

Code spec for CallSimpleThread.java:

```
public class CallSimpleThread
{
    public static void main (String[] args)
    {
        new SimpleThread("Main Thread").start();
        new SimpleThread("Child Thread").start();
    }
}
```

Explanation:
Two new instances of the SimpleThread.java file is created, which is passed the name of the thread. The start() method inherited from the Thread class is invoked on the object of the class to make the thread eligible for running.

Once the .java file is ready, it needs to be compiled by the Java compiler [javac] and then the .class file is interpreted by the Java interpreter [java]. After compiling and executing the window as shown in diagram 30.3 appears.

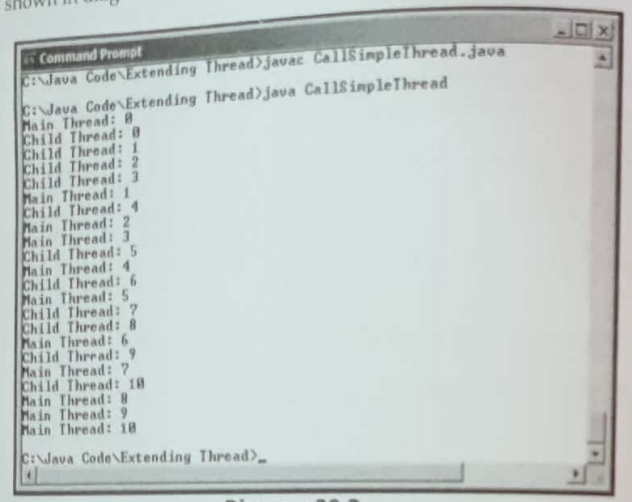

Diagram 30.3

Choosing An Approach

The Thread class contains multiple methods that can be overridden by a derived class which will determine the functionality of the derived class.

Of all the methods contained within the Thread class, the only method that **must** be overridden is **run()**. This is same method required when **Runnable** has to be implemented.

Many Java programmers feel that a class should be extended only when it is being enhanced or modified in someway. With this in mind, it's sensible to implement **Runnable** if no other methods of the Thread class are being overridden, when implementing a Thread.

Chapter 31

SECTION VI: CLIENT SIDE PROGRAMMING

Sockets And Networking

In today's computing world there are many expensive resources such as Laser printers, Fax machines and so on that need to be shared. Business data is another expensive resource that needs to be shared. To facilitate this, robust networks have come into existence. Networks allow expensive resources to be shared.

Any programming environment must provide standard techniques to permit its applications to communicate across a network. Java is a programming environment and it provides simple yet robust techniques that permit Java applications to communicate across networks and share valuable resources.

Java communication is built on standard Client/Server architecture. A Server must have a port number on which some software is a listener/talker. What this really means is that all Servers have software listening on a port determined by the hardware architecture of the computer.