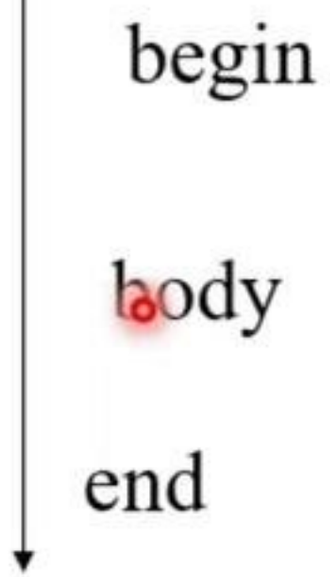
Multi-threading programming

A single threaded program

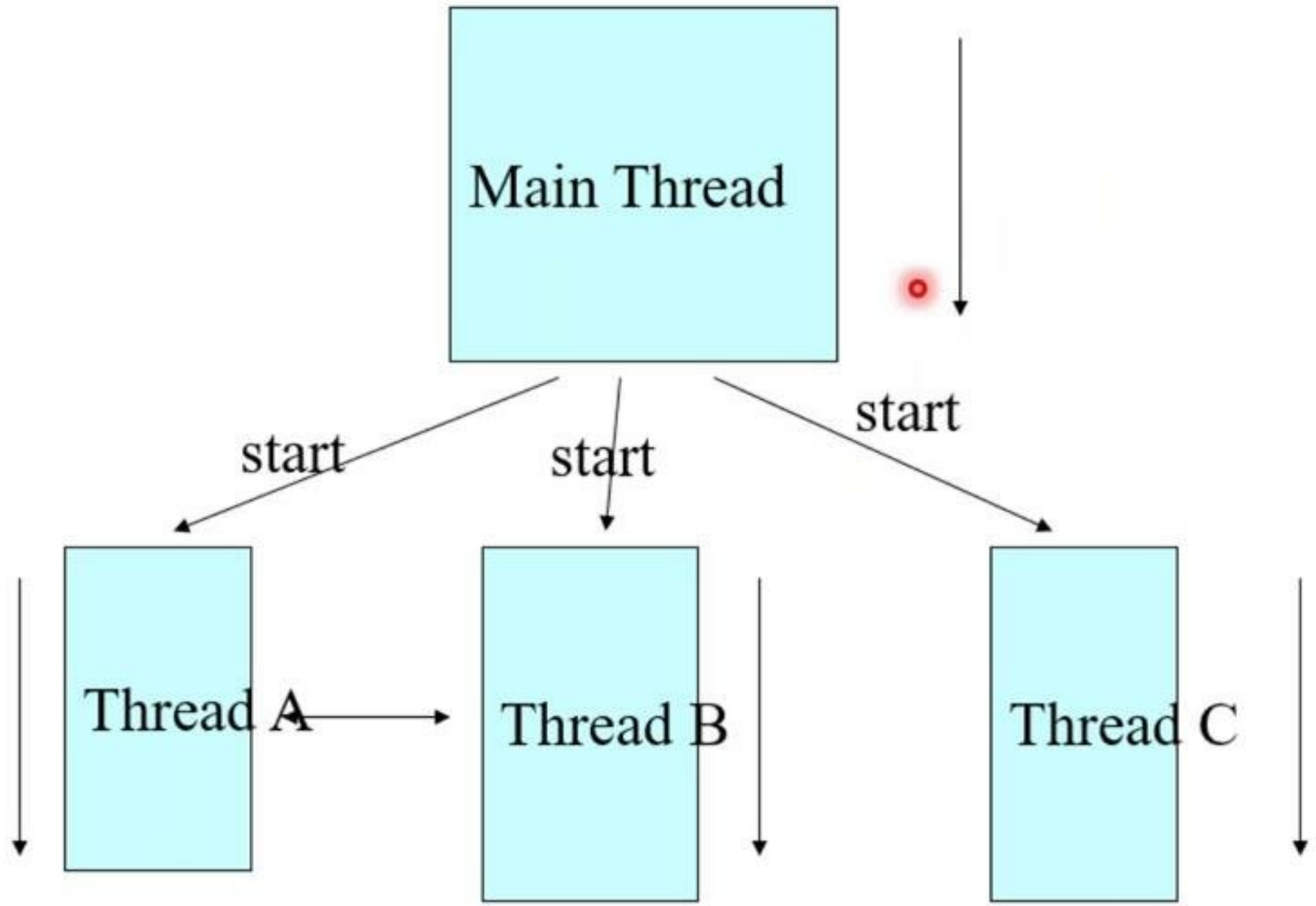

```
class ABC
{
....
    public void main(..)
    {
    ...
    ..
    }
}
```



Multitasking Vs Multithreading

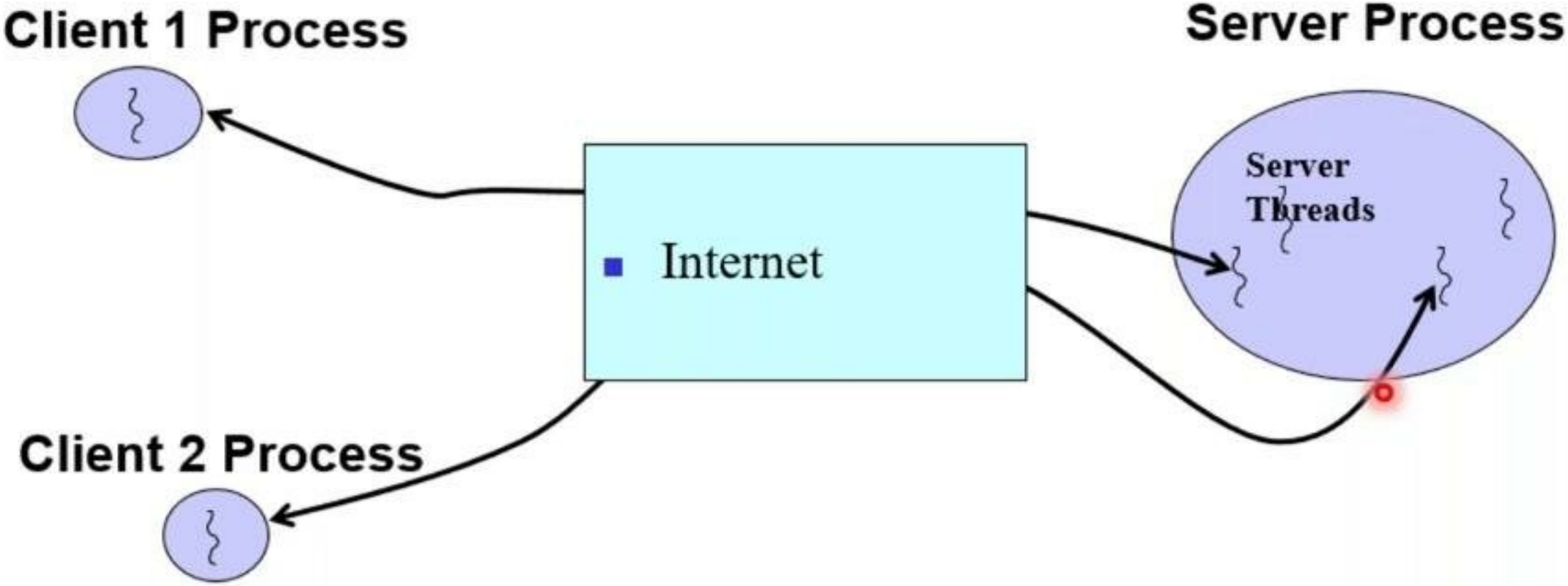
- `Multitasking` is the ability to run one or more programs concurrently.
- Operating system controls the way in which these programs run by scheduling them.
- Time elapsed between switching of programs is minuscule.
- `Multithreading` is the ability to execute different parts of a program, called `threads`, simultaneously.

A Multithreaded Program



Threads may switch or exchange data/results

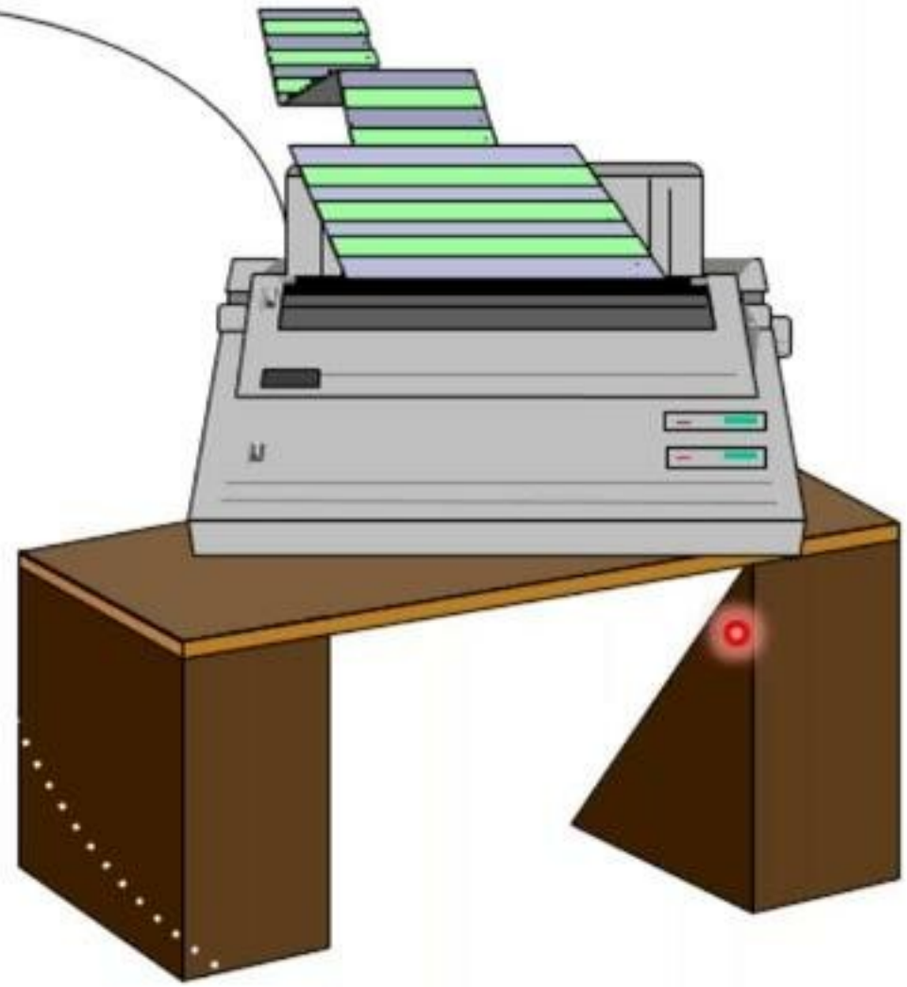
Multithreaded Server



Modern Applications need Threads

Printing Thread

Editing Thread



What are Threads Good For?

- To maintain responsiveness of an application during a long running task

What are Threads Good For?

- To maintain responsiveness of an application during a long running task
- To enable cancellation of separable tasks

Benefits of Multithreading

- Multithreading requires less overhead than multitasking.
  - In multitasking, processes run in their own different address space.
  - Tasks involved in multithreading can share the same address space.
- Inter-process calling involves more overhead than inter-thread communication.
- Multithreading allows us to write efficient programs that make maximum use of the CPU.
- Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause.

The 'main' thread

- When Java programs execute, there is always one thread running and that is the main thread.
  - It is the thread from which child threads are created.
  - Program is terminated when main thread stops execution.
  - Main thread can be controlled through `Thread` objects.

Application Thread

- When we execute an application:

Application Thread

- When we execute an application:
  1. The JVM creates a Thread object whose task is defined by the **`main()`** method

Application Thread

- When we execute an application:
  1. The JVM creates a Thread object whose task is defined by the **`main()`** method
  2. The JVM starts the thread
  3. The thread executes the statements of the program one by one
  4. After executing all the statements, the method returns and the thread dies

Creating Threads

- There are two ways to create our own Thread object
  - Subclassing the **Thread** class and instantiating a new object of that class
  - Implementing the **Runnable** interface
- In both cases the run() method should be implemented.

Creating Threads by extending Thread class

Example 1 –

Creating a thread by extending the `Thread` class

```
class MyThread extends Thread
{
  public static void main(String args[])
  {
    MyThread Objex = new MyThread();
    Objex.create();
    System.out.println("This is the main thread");
  }

  public void create()
  {
      Thread Objth = new Thread(this);
    Objth.start();
   }
```

Creating Threads by extending Thread class

Example 1 –

Creating a thread by extending the `Thread` class

```
public void run()
{
    while(true)
    {
        try
        {
            System.out.println("This is the child thread");
          Thread.sleep(500);
            }
            catch(InterruptedException e)
        { }
    }
  }
}
```

Implementing Runnable


```
public class RunnableExample implements
  Runnable {

  public void run () {
      for (int i = 1; i <= 100; i++) {
              System.out.println ("***");
          }
      }
}
```

Starting the Threads


```
public class ThreadsStartExample {
        public static void main (String argv[])
  {
            new ThreadExample ().start ();
           new Thread(new RunnableExample
  ()).start ();
        }
}
```

Thread States 3-2

- **Running:** Thread enters the running state when it starts executing.

- **Sleeping:** Execution of a thread can be halted temporarily by using sleep() method. The thread becomes ready after sleep time expires.

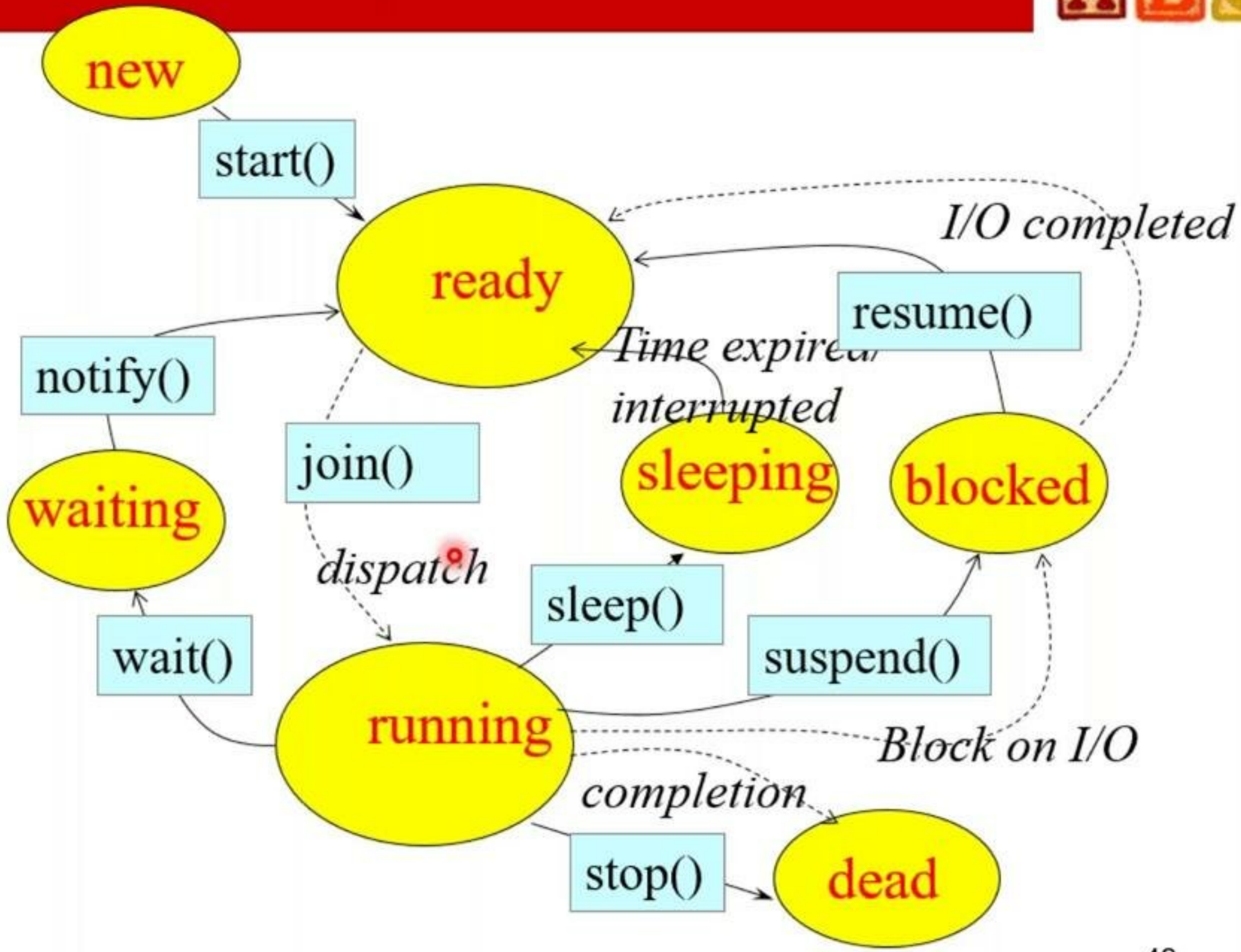
Example


```
public class PrintThread1 extends Thread {
    String name;
    public PrintThread1(String name) {
        this.name = name;
    }
    public void run() {
        for (int i=1; i<100 ; i++) {
            try {
                sleep((long)(Math.random() * 100));
            } catch (InterruptedException ie) { }
            System.out.print(name);
        }
}
```

Demonstration: Example 3

Threads.java (Threads.java.JAV)

Life Cycle of Thread

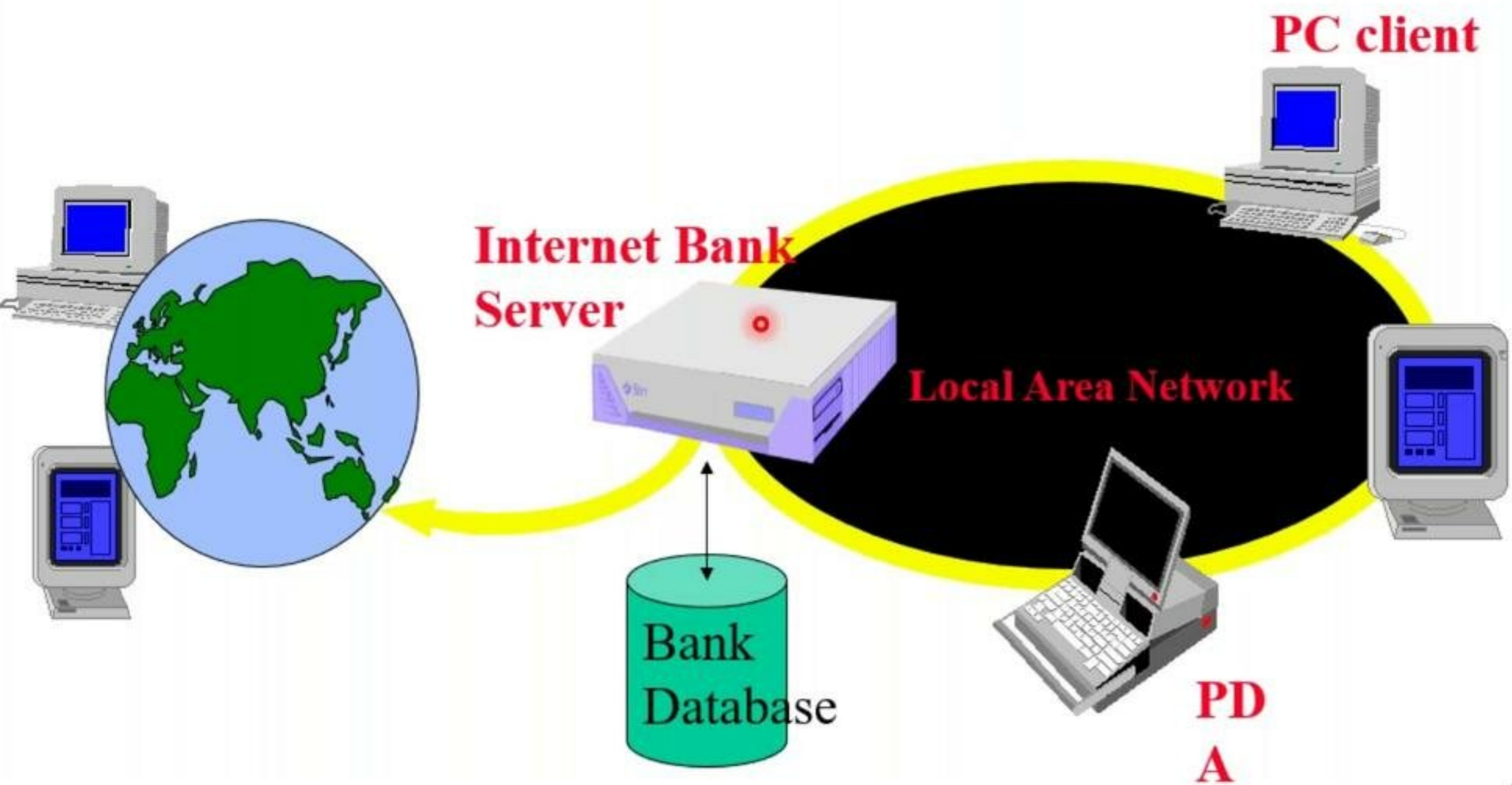


Thread Priority Example

Demonstration: Example 5

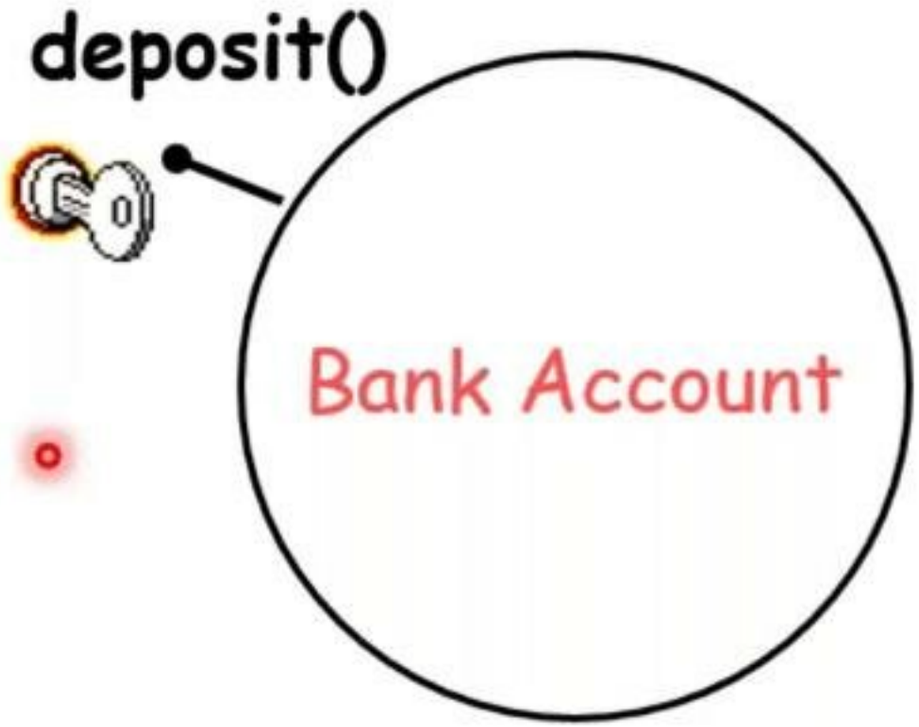
ThreadPriority.java (ThreadPriority.java.JAV)

Online Bank: Serving Many Customers and Operations



t3 t2 t1

Critical Sections



Using ‘wait-notify’ mechanism

- Java provides a well-designed inter-thread communication mechanism using the `wait()`, `notify()` and `notifyAll()` methods.
- The methods are implemented as `final` methods in the class `Object`.
- `wait()`, `notify()` and `notifyAll()` can be called only from within a `synchronized` method.

wait()

- Points to remember while using the `wait()` method:
  - The calling thread gives up the CPU.
  - The calling thread gives up the lock.
  - The calling thread goes into the waiting pool of the monitor.

Consumer


```
synchronized (lock) {
    while (!resourceAvailable()) {
        lock.wait();
    }
    consumeResource();
}
```

DXC - 00010.Multithreading/src/extendingthread/ThreadDemo.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer

- 0000000.planning
- 00001.Greetings
- 00002.ems
- 00003.ArrayInJava
- 00004.Inheritance
- 00005.AbstractClasss
- 00006.Interfaces
- 00007.ExceptionHandling
- 00008.IOperation
- 00009.Collection
- 00010.Multithreading
  - JRE System Library [jre]
  - src
    - extendingthread
      - ThreadDemo.java

*ThreadDemo.java

```
10              } catch (InterruptedException e) {
11                  // TODO Auto-generated catch block
12                  e.printStackTrace();
13              }
14          }
15          System.out.print("\n thread:"+this.getName()+" is out...");
16      }
17  }
18
19  public class ThreadDemo2 extends Thread{
20      public void run() {
21          System.out.print("\n got processor time..");
22          for(int i=1;i<30;i++) {
23              System.out.print("\nThread name:"+this.getName()+" Thread Priority:"+this.getPriority());
24              if(i==20) {
25                  this.get
26              }
27          }
28          System.out.pri
```

getClass() : Class<?> - Object
getContextClassLoader() : ClassLoader - Thread
getId() : long - Thread
getName() : String - Thread
getPriority() : int - Thread
getStackTrace() : StackTraceElement[] - Thread
getState() : State - Thread
getThreadGroup() : ThreadGroup - Thread
getUncaughtExceptionHandler() : UncaughtExceptionHandler -
getAllStackTraces() : Map<Thread,StackTraceElement[]> - Thre

Press 'Ctrl+Space' to show Template Proposals

Returns the runtime class of this `Object`. The returned `Class` object is the object that is locked by `static synchronized` methods of the represented class.

The actual result type is `Class<? extends |X|>` where `|X|` is the erasure of the static type of the expression on which `getClass` is called. For example, no cast is required in this code fragment:

```
Number n = 0;
Class<? extends Number> c = n.getClass();
```

Returns:

The `Class` object that represents the runtime class of this object.

@jls

Press 'Tab' from proposal table or click for focus

Problems Javadoc Declaration

No consoles to display at this time.

Writable Smart Insert 25 : 25 : 742

DXC - 00010.Multithreading/src/extendingthread/ThreadDemo.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer

- 0000000.planning
- 00001.Greetings
- 00002.ems
- 00003.ArrayInJava
- 00004.Inheritance
- 00005.AbstractClasss
- 00006.Interfaces
- 00007.ExceptionHandling
- 00008.IOperation
- 00009.Collection
- 00010.Multithreading
  - JRE System Library [jre]
  - src
    - extendingthread
      - ThreadDemo.java

*ThreadDemo.java

```
10              } catch (InterruptedException e) {
11                  // TODO Auto-generated catch block
12                  e.printStackTrace();
13              }
14          }
15          System.out.print("\n thread:"+this.getName()+" is out...");
16      }
17  }
18
19  public class ThreadDemo2 extends Thread{
20      public void run() {
21          System.out.print("\n got processor time..");
22          for(int i=1;i<30;i++) {
23              System.out.print("\nThread name:"+this.getName()+" Thread Priority:"+this.getPriority());
24              if(i==20) {
25                  System.out.print("\nExisting priority is:"+this.getPriority()+ "New priority:"+this.set
26              }
27          }
28          System.out.print("\n thr
```

Sets the context ClassLoader for this Thread. The context ClassLoader can be set when a thread is created, and allows the creator of the thread to provide the appropriate class loader, through `getContextClassLoader`, to code running in the thread when loading classes and resources.

If a security manager is present, its `checkPermission` method is invoked with a `RuntimePermission("setContextClassLoader")` permission to see if setting the context ClassLoader is permitted.

Parameters:

cl the context ClassLoader for this Thread, or null indicating the system class loader (or, failing that, the bootstrap class loader)

Throws:

Press 'Tab' from proposal table or click for focus

- setContextClassLoader(ClassLoader cl) : void - Thread
- setDaemon(boolean on) : void - Thread
- setName(String name) : void - Thread
- setPriority(int newPriority) : void - Thread
- setUncaughtExceptionHandler(UncaughtExceptionHandler eh) : vo
- setDefaultUncaughtExceptionHandler(UncaughtExceptionHandler

Press 'Ctrl+Space' to show Template Proposals

Problems Javadoc Declaration Console

No consoles to display at this time.

Writable Smart Insert 25 : 104 : 821