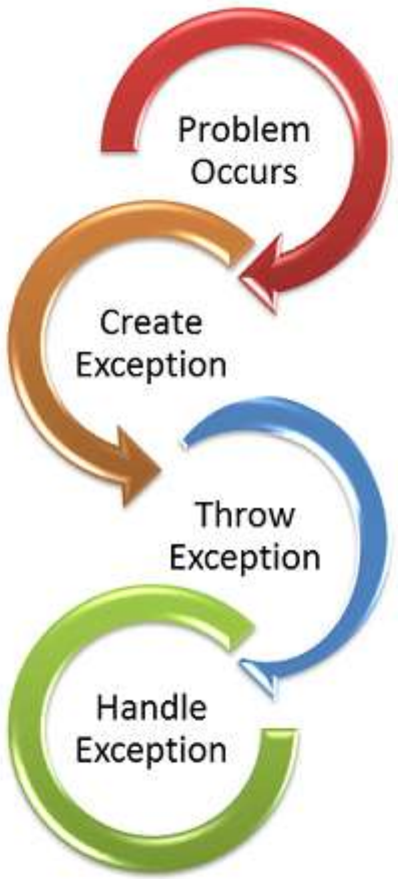


Exception Handling

Errors

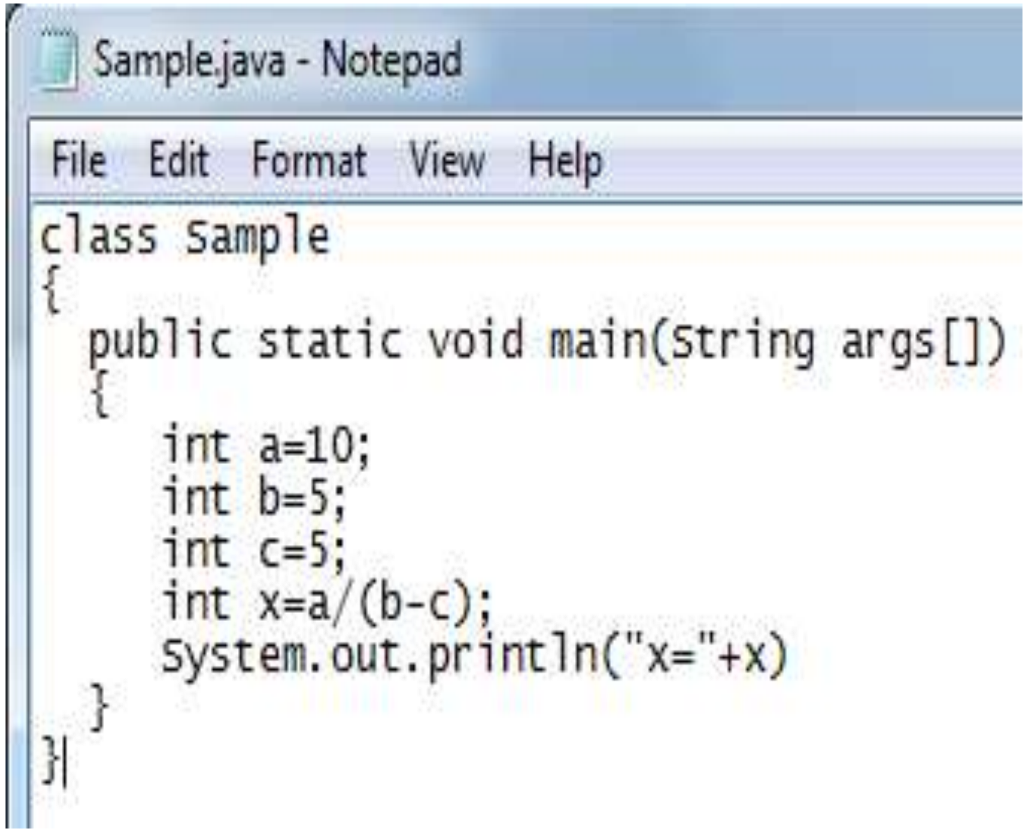
- An error may produce an incorrect output or may terminate the execution of the program abruptly or even may cause the system to crash.

Types of errors

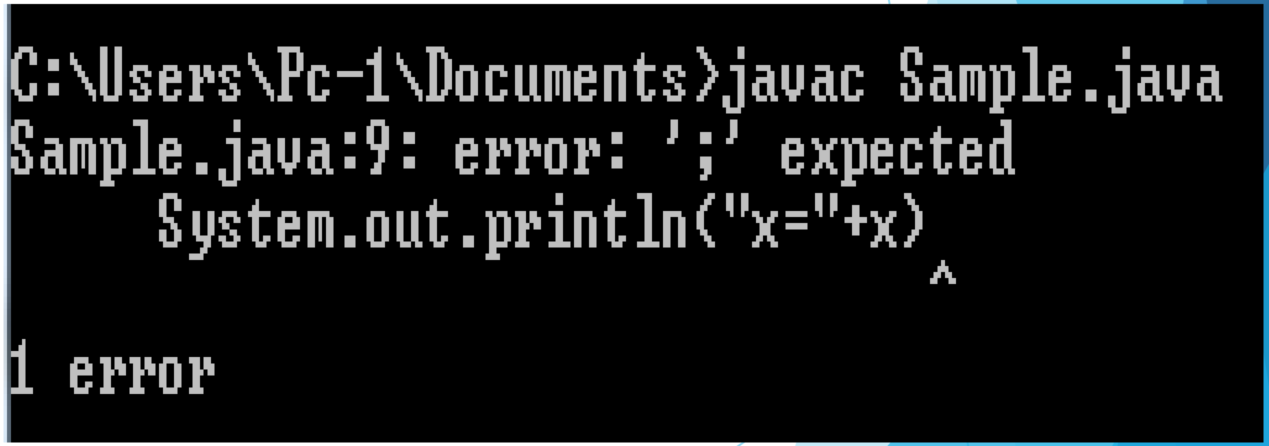
- Compile time Error
- Runtime Error

Compile Time Error

- All syntax errors will be detected and displayed by the java compiler and therefore these errors are known as compile time errors.
- Whenever the compiler displays an error, it will not create the .*class* file.



```
class Sample
{
  public static void main(String args[])
  {
     int a=10;
     int b=5;
     int c=5;
     int x=a/(b-c);
     System.out.println("x="+x)
  }
}
```



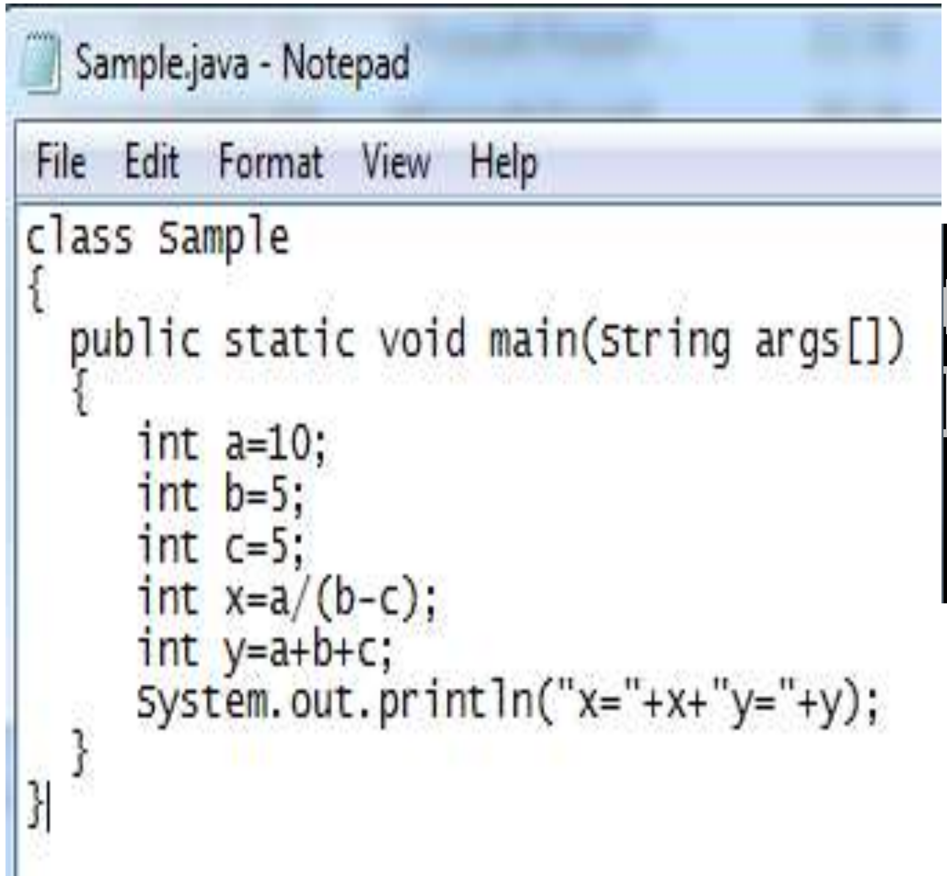
```
C:\Users\Pc-1\Documents>javac Sample.java
Sample.java:9: error: ';' expected
     System.out.println("x="+x)
                               ^

1 error
```

Most common compile time errors

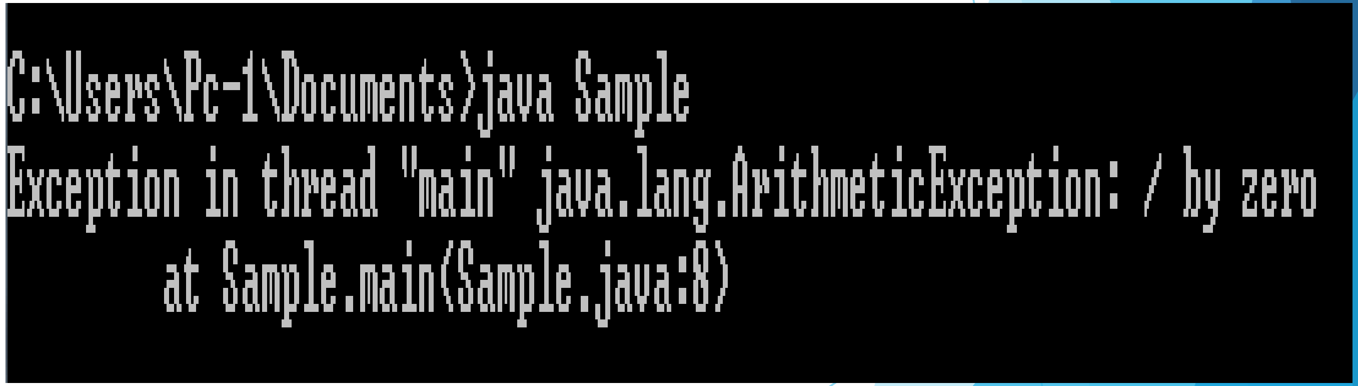
- Missing semicolons
- Missing brackets in classes and methods
- Misspelling identifiers and keywords
- Missing double quotes in strings
- Use of undeclared variables
- Use of = in place of == operator

Runtime Error

- Sometimes, a program may compile successfully creating the .*class* file but may not run properly.
- Such programs may produce wrong results due to wrong logic.



```
class Sample
{
  public static void main(String args[])
  {
      int a=10;
      int b=5;
      int c=5;
      int x=a/(b-c);
      int y=a+b+c;
      System.out.println("x="+x+"y="+y);
  }
}
```



```
C:\Users\Pc-1\Documents>java Sample
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Sample.main(Sample.java:8)
```

Most common run time errors

- Dividing an integer by zero
- Accessing an element that is out of the bound of an array
- Attempting to use a negative size for an array
- Accessing a character that is out of bounds of a string
- Trying to store a value into an array of incompatible type

Exceptions

- An exception is a condition that is caused by a runtime error in the program.

- When the java interpreter encounters an error such as dividing an integer by zero, it creates an exception object and throws it(i.e. informs us that an error has occurred)

Exception Handling

- If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions, this task is known as *Exception Handling*.

- Task performs during Exception Handling:
  - Find the problem (*Hit* the exception)
  - Inform that an error has occurred (*Throw* the exception)
  - Receive the error information (*Catch* the exception)
  - Take corrective actions (*Handle* the exception)

Exception Handling Mechanism

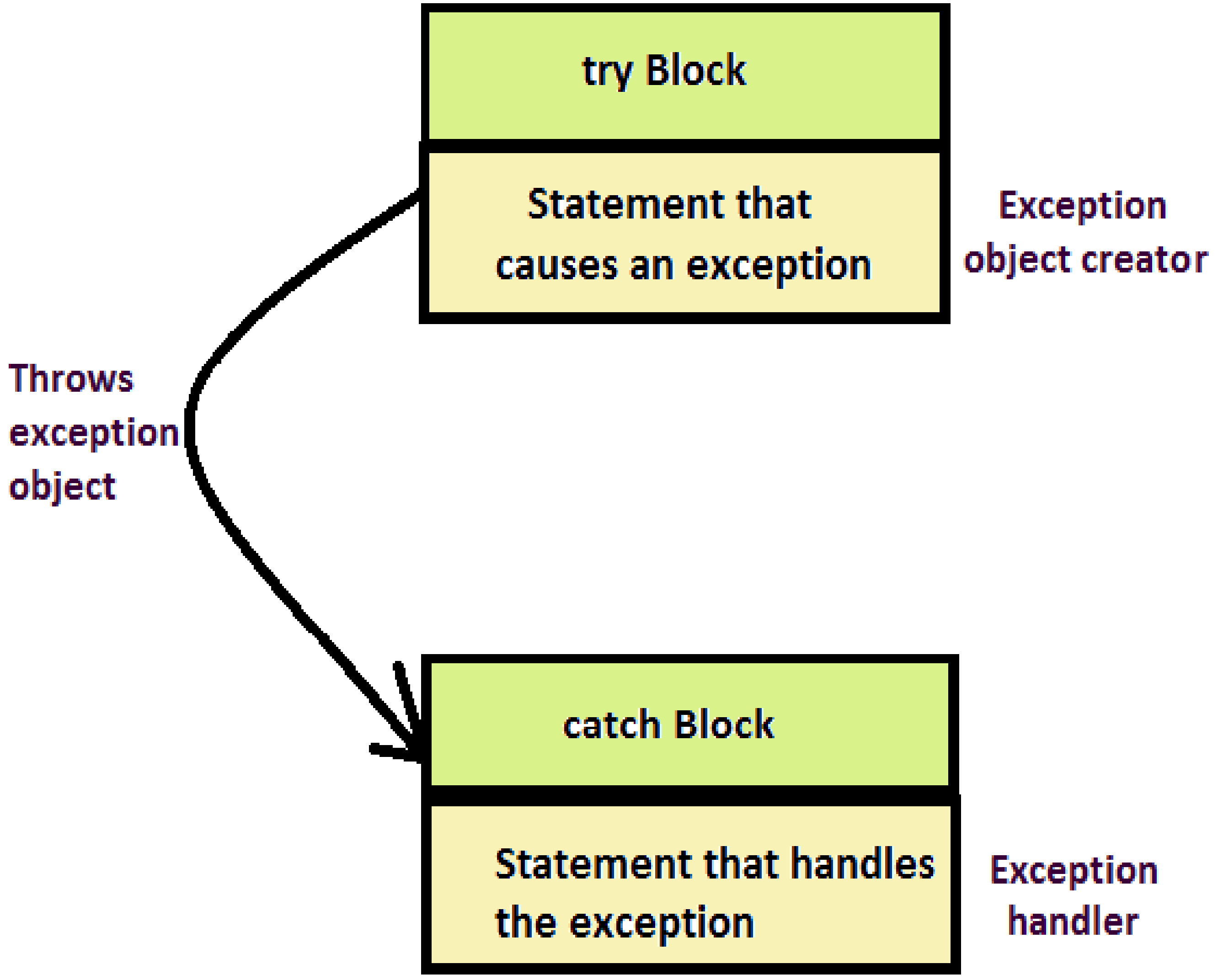


Fig: -Exception handling mechanism

Syntax of exception handling code

```
………
try
{
      statement;        //generates an exception
}
 catch(Exception_type e)
 {
      statement;        //processes the exception
 }
  ……….
  …………
```

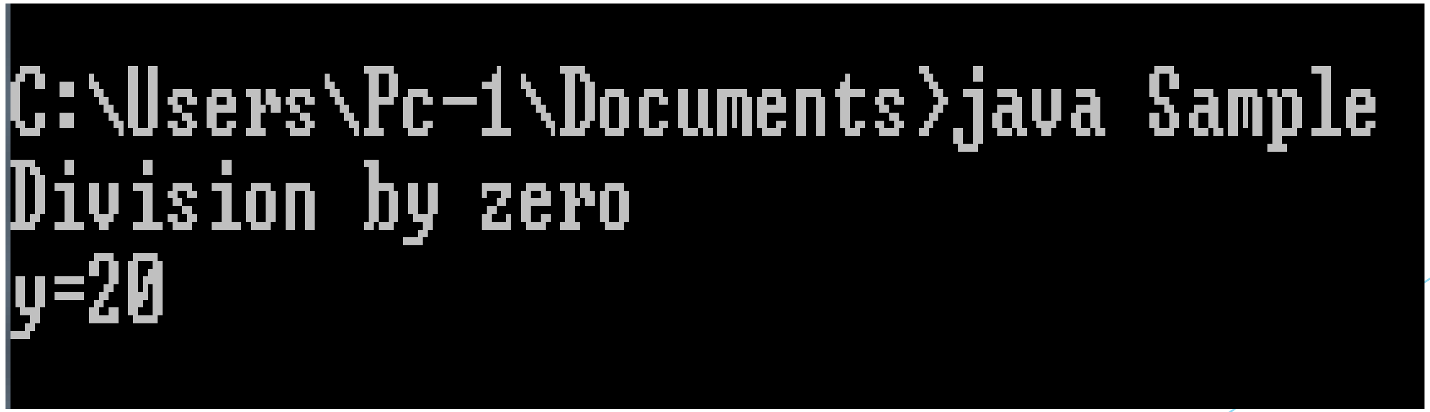
Example

```
class Sample
{
 public static void main(String args[])
 {
    int a=10;
    int b=5;
    int c=5;
    int x=0,y=0;
    try
    {
         x=a/(b-c);
         System.out.println("x="+x);
     }
```

Contd…

Contd...

```
catch(ArithmeticException e)
    {
        System.out.println("Division by zero");
     }
     y=a+b+c;
    System.out.println("y="+y);
  }
}
```



```
C:\Users\Pc-1\Documents>java Sample
Division by zero
y=20
```

Common java exceptions

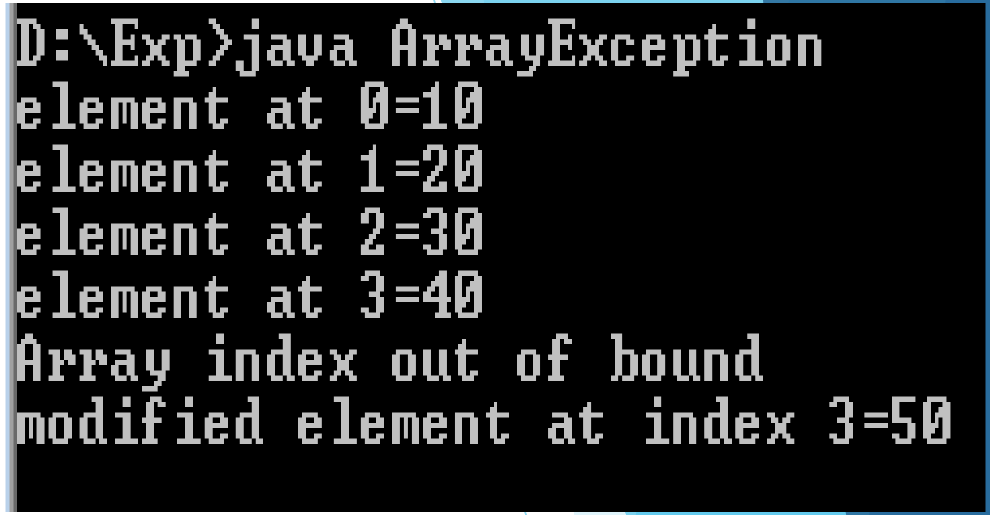
Exception Type	Cause of Exception
ArithmaticException	Caused by math errors such as division by zero
ArrayIndexOutOfBoundsException	Caused by array indexes
ArrayStoreException	Caused when a program tries to store the wrong type of data in an array
FileNotFoundException	Caused by an attempt to access a nonexistent file
IOException	Caused by general I/O failure, such as inability to read from a file

Question

Write a java program to show Array Index out of Bounds Exception.

Solution

```
class ArrayException
{
 public static void main(String args[])
 {
  int  A[]={10,20,30,40};
  try
  {
   for(int i=0;i<5;i++)
           System.out.println("element at "+i+"="+A[i]);
   }
  catch(ArrayIndexOutOfBoundsException e)
  {
       System.out.println("Array index out of bound");
   }
  A[3]=50;
  System.out.println("modified element at index 3="+A[3]);
 }
}
```



```
D:\Exp>java ArrayException
element at 0=10
element at 1=20
element at 2=30
element at 3=40
Array index out of bound
modified element at index 3=50
```

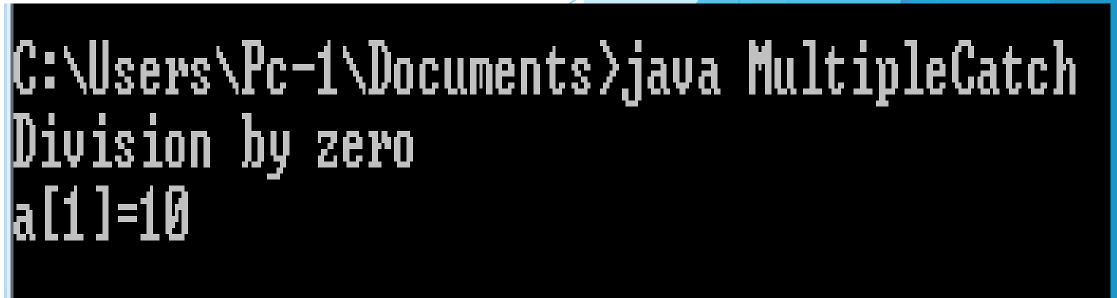
Multiple catch statements

Syntax :

```
try
{
    statement;
}
catch(Exception_type-1 e)
{   statement ;     }
catch(Exception_type-2 e)
{   statement ;     }
catch(Exception_type-n e)
{   statement ;     }
```

Example

```
public class MultipleCatch
{
   public static void main(String args[])
   {
       int a[]=new int[5];
       a[1]=10;
      try
      {
          a[5]=30/0;
      }
      catch(ArithmeticException e)
     {          System.out.println("Division by zero");           }
     catch(ArrayIndexOutOfBoundsException e)
     {          System.out.println("Array Index out of bound");   }
     System.out.println("a[1]="+a[1]);
  }
}
```



```
C:\Users\Pc-1\Documents>java MultipleCatch
Division by zero
a[1]=10
```

Example

- Accept 2 integer numbers from command line.
- Divide first number by second and print the result.
- Show ArithmeticException and NumberFormatException using multiple catch statement.

Solution

```
class Multicatch
{
 public static void main(String args[])
 {
  try
  {
   int n=Integer.parseInt(args[0]);
   int n1=Integer.parseInt(args[1]);
   int n2=n/n1;
   System.out.println("n2= "+n2);
  }
  catch(ArithmeticException e)
  {      System.out.println("Arithmatic exception");    }
  catch(NumberFormatException e)
  {      System.out.println("number format exception occurred. Enter integer numbers");    }
 }
}
```

Nested try block

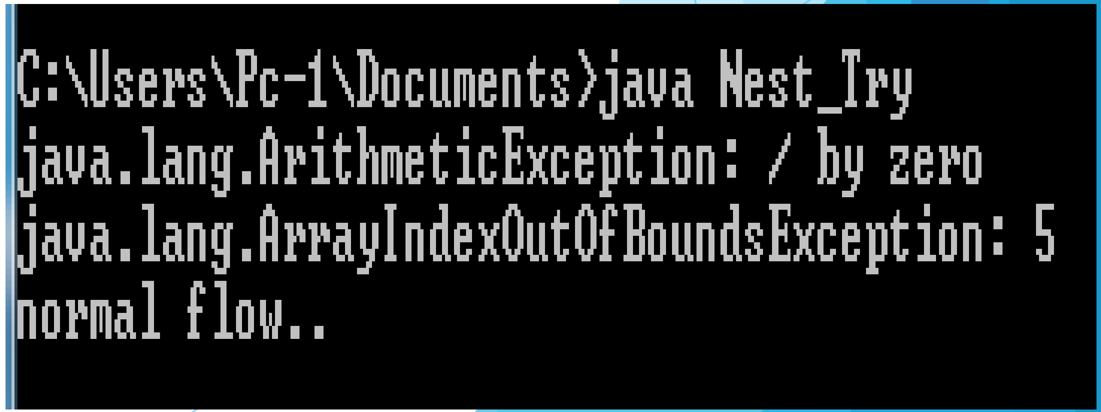
- The try block within a try block is known as nested try block in java.
- Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error.
- In such cases, exception handlers have to be nested.

Syntax

```
try
{
        statement 1;
        statement 2;
        try
       {
            statement 1;
            statement 2;
        }
        catch(Exception e)
        {  ……………
        }
}
catch(Exception e)
{  ……………..
}
```

Example

```
class Nest_Try
{
 public static void main(String args[])
 {
  try
  {
       try
       {
            int b =39/0;
        }
        catch(ArithmeticException e)
        {      System.out.println(e);    }
        int a[]=new int[5];
        a[5]=4;
  }
  catch(ArrayIndexOutOfBoundsException e)
  {      System.out.println(e);      }
  System.out.println("normal flow..");
 } }
```



```
C:\Users\Pc-1\Documents>java Nest_Try
java.lang.ArithmeticException: / by zero
java.lang.ArrayIndexOutOfBoundsException: 5
normal flow..
```

Finally statement

- Finally statement can be used to handle an exception that is not caught by any of the previous catch statement
- Finally block can be used to handle any exception generated within a try block.
- Syntax :

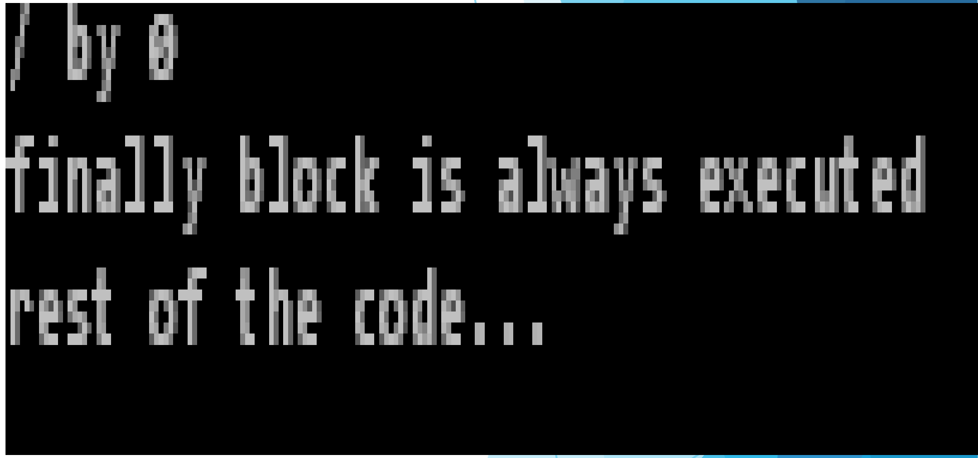
```
try
{
        ……..
}
finally
{
        ………
}
```

OR

```
try
{
        ………
}
catch(..)
{
        ………
}
finally
{    …….      }
```

Example

```
 class TestFinally
{
  public static void main(String args[])
 {
  try
  {
      int data=25/0;
      System.out.println(data);
  }
  catch(ArithmeticException e)
  {     System.out.println("/ by 0");       }
  finally
 {         System.out.println("finally block is always executed");          }
  System.out.println("rest of the code...");
  }
}
```



```
/ by 0
finally block is always executed
rest of the code...
```

Throwing our Own Exception

- A programmer can create a new exception and throw it explicitly.
- These exceptions are known as user-defined exceptions.
- In order to throw user defined exceptions, ***throw*** keyword is being used.

Syntax :

```
throw new Throwable_subclass;
```

Example :

```
throw new ArithmeticException();
```

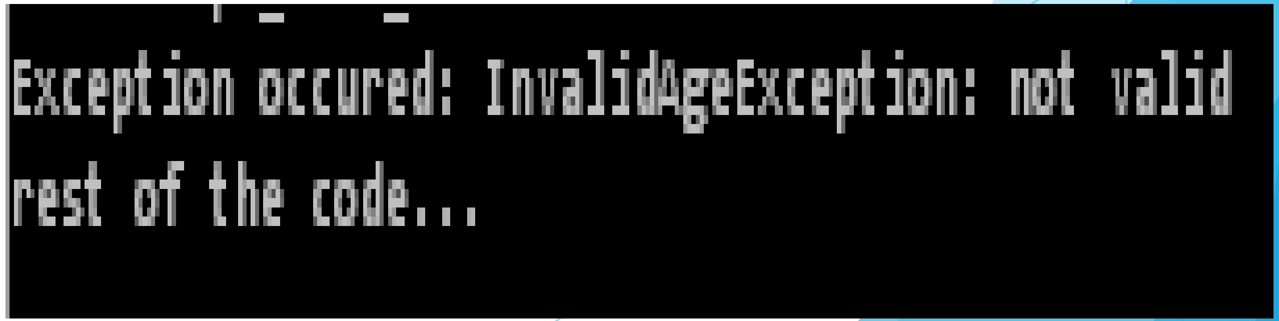
Example

```
class InvalidAgeException extends Exception
{
    InvalidAgeException(String s)
   {
        super(s);
    }
}
class TestThrow
{
    static void validate(int age)throws InvalidAgeException
   {
     if(age<18)
            throw new InvalidAgeException("not valid");
     else
            System.out.println("welcome to vote");
   }
```

Contd…

Contd…

```
public static void main(String args[])
{
    try
   {
       validate(13);
    }
    catch(Exception m)
    {
        System.out.println("Exception occured: "+m);
    }
      System.out.println("rest of the code...");
 }
}
```



```
Exception occured: InvalidAgeException: not valid
rest of the code...
```

throws keyword

- The **throws keyword** is used in method declaration, in order to explicitly specify the exceptions that a particular method might throw.
- When a method declaration has one or more exceptions defined using throws clause then the method-call must handle all the defined exceptions.
- Syntax :

```
    return_type method_name() throws exception_class_name
    {     //method code   }
```

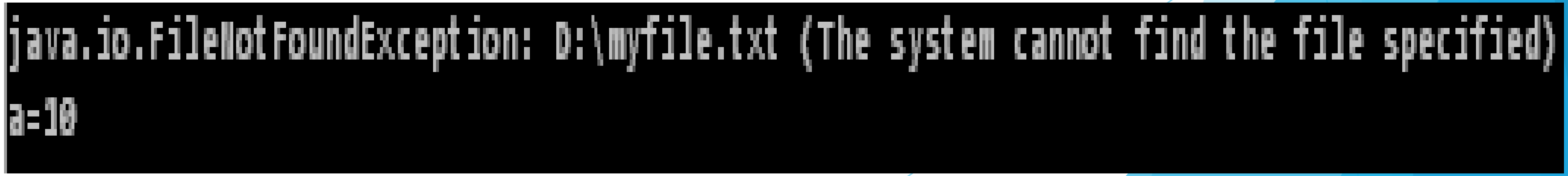
Example

```
import java.io.*;
class FileExThrow
{
    void filecall() throws FileNotFoundException
   {
       FileInputStream fis = new FileInputStream("D:/myfile.txt");
    }
}
class Test_Throws
{
  public static void main(String args[])
  {
    int a=10;
```

Contd…

Contd…

```
  try
  {
    FileExThrow obj=new FileExThrow();
    obj.filecall();
  }
  catch(Exception e)
  {
     System.out.println(e);
  }
  System.out.println("a="+a);
  }
}
```

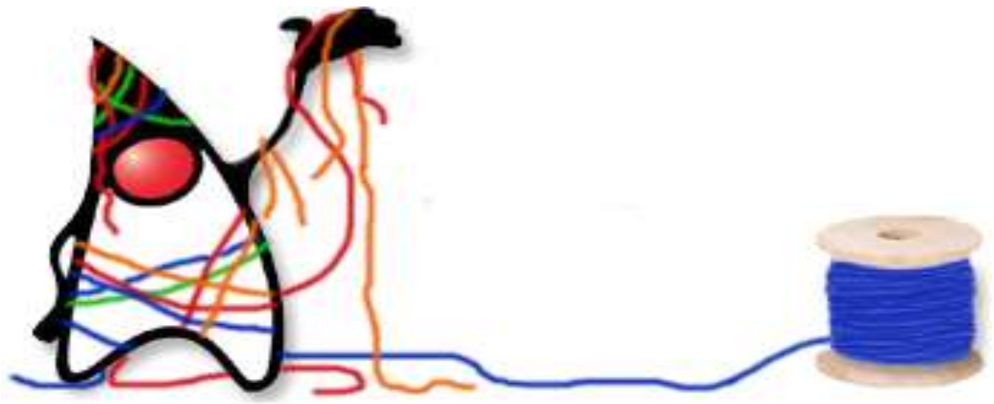


```
java.io.FileNotFoundException: D:\myfile.txt (The system cannot find the file specified)
a=10
```

Example

```
import java.io.*;
 class ThrowsExample {
  void mymethod(int num)throws IOException, ClassNotFoundException{
     if(num==1)
        throw new IOException("Exception Message1");
     else
        throw new ClassNotFoundException("Exception Message2");
  } }
class TestThrows{
  public static void main(String args[]){
   try{
     ThrowsExample obj=new ThrowsExample();
     obj.mymethod(1);
   }catch(Exception ex){
     System.out.println(ex);
    } }}
```

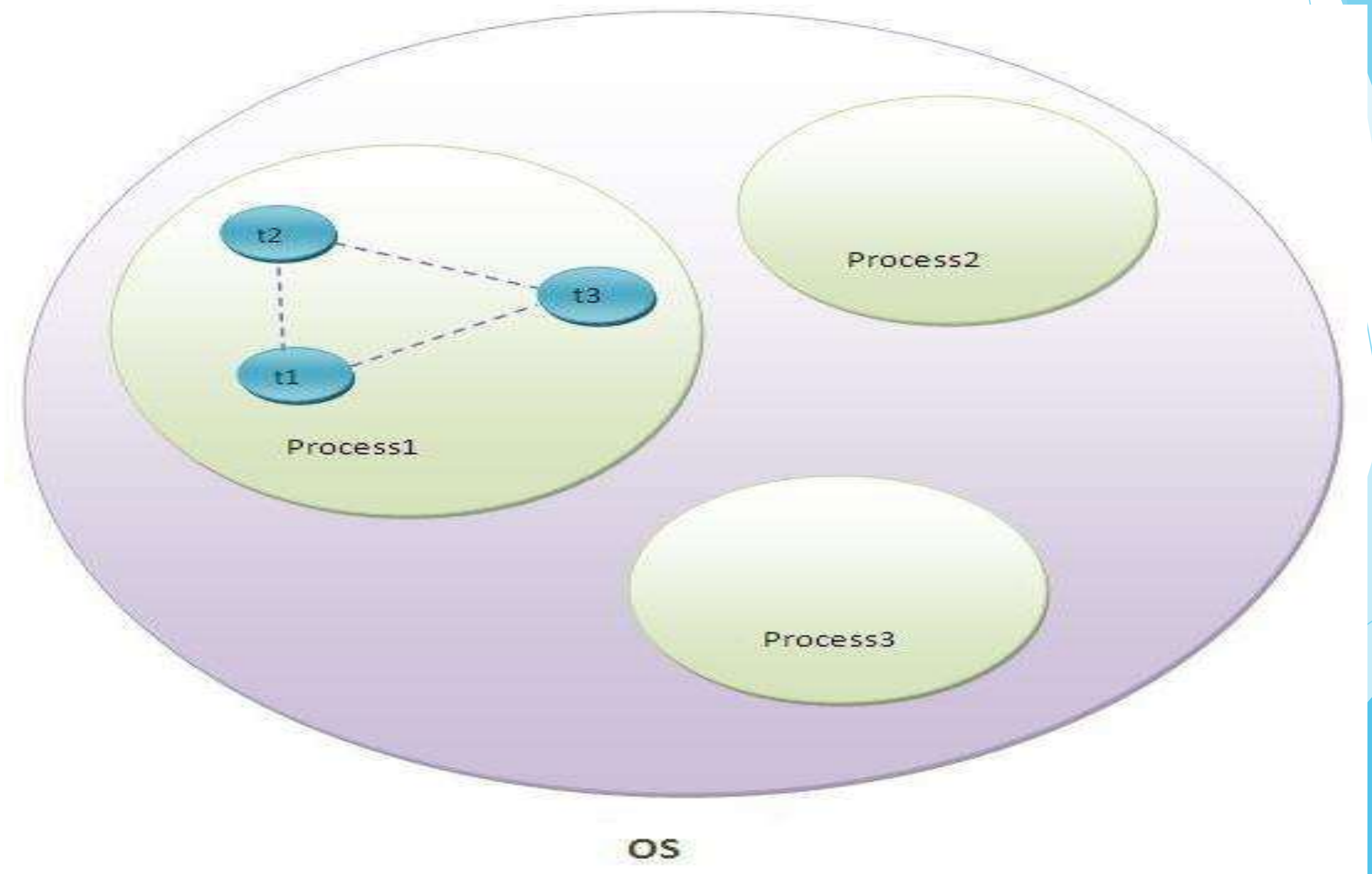
Multithread programming



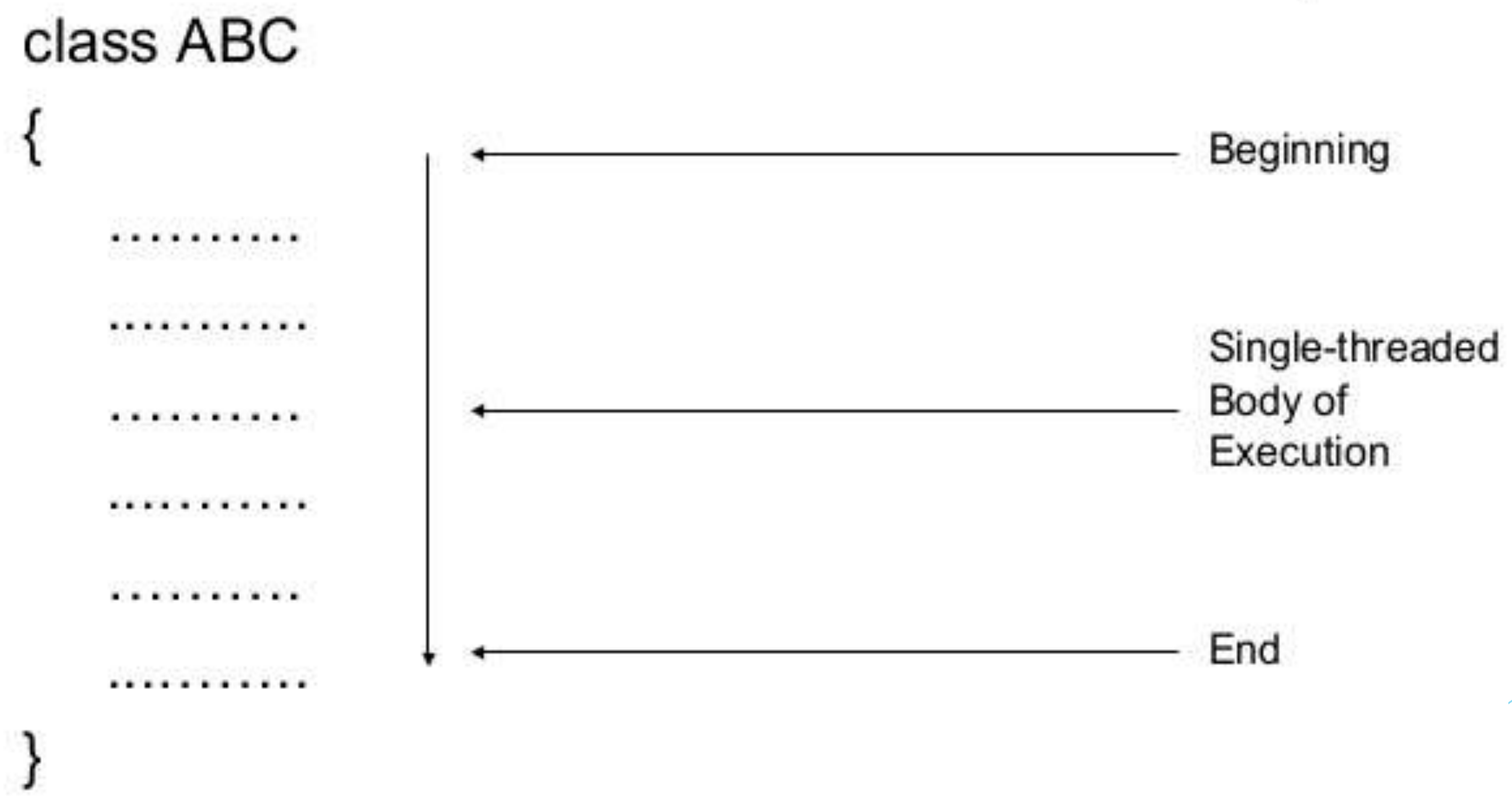
Threads

- A thread is a light-weight smallest part of a process that can run concurrently with the other parts(other threads) of the same process.
- Threads are independent because they all have separate path of execution that’s the reason if an exception occurs in one thread, it doesn’t affect the execution of other threads.
- All threads of a process share the common memory.

Thread in java



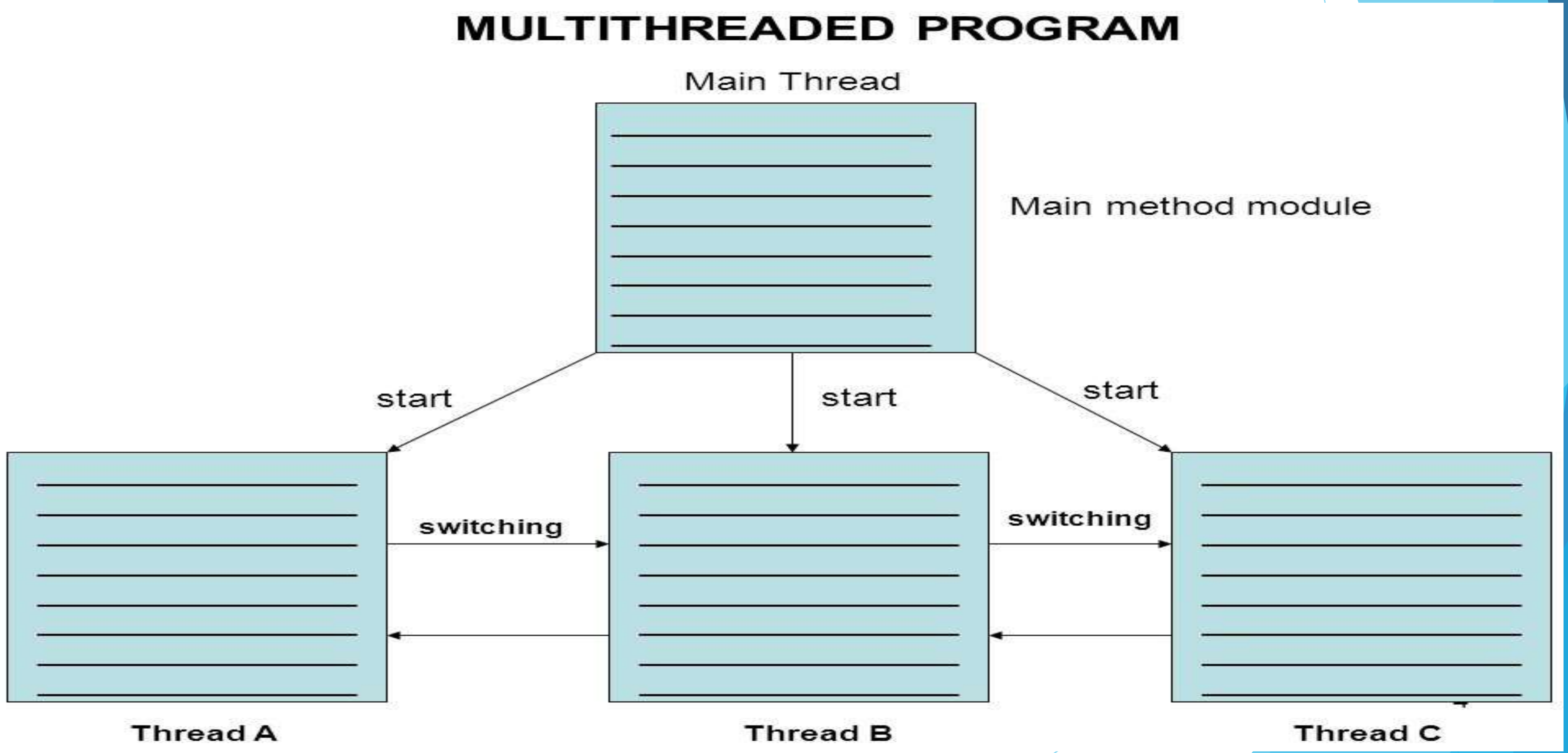
Single threaded program



Multithreading

- The process of executing multiple threads simultaneously is known as multithreading.
- A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task.

A multithreaded program



Creating a Thread

- Thread are implemented in the form of objects that contain a method called run().
- It makes up the entire body of a thread and is the only method in which thread's behavior can be implemented.
- Syntax:

```
public void run()
{
    ….//statement for implementing thread
}
```

Ways to create Thread

- **By extending the Thread class**

  define a class that extends Thread class and override its run() method with the code required by the thread

- **By implementing Runnable interface**

  define a class that implements Runnable interface. The Runnable interface has only one method, run().

Extending the Thread class

Steps to create a Thread by extending Thread class:

1. Declare the class as extending the Thread class.
2. Implement the run() method that is responsible for executing the sequence of code that the thread will execute.
3. Create a thread object and call the start() method to initiate the thread execution.

1. Declaring the class

Syntax:

```
    class classname extends Thread
    {   ...........}
```

Example

```
    class myThread extends Thread
      {  ..........}
```

2. Implementing the run() method

- The run() method has been inherited by the subclass which we created previously.
- We have to override this method in order to implement the code to be executed by our thread.
- Syntax :

```
public void run()
{
    ……//Thread code
}
```

3. Starting new Thread

Syntax:

```
classname objectname=new classname();
objectname.start();
```

Example :

```
MyThread t=new MyThread();
t.start();
```

Example

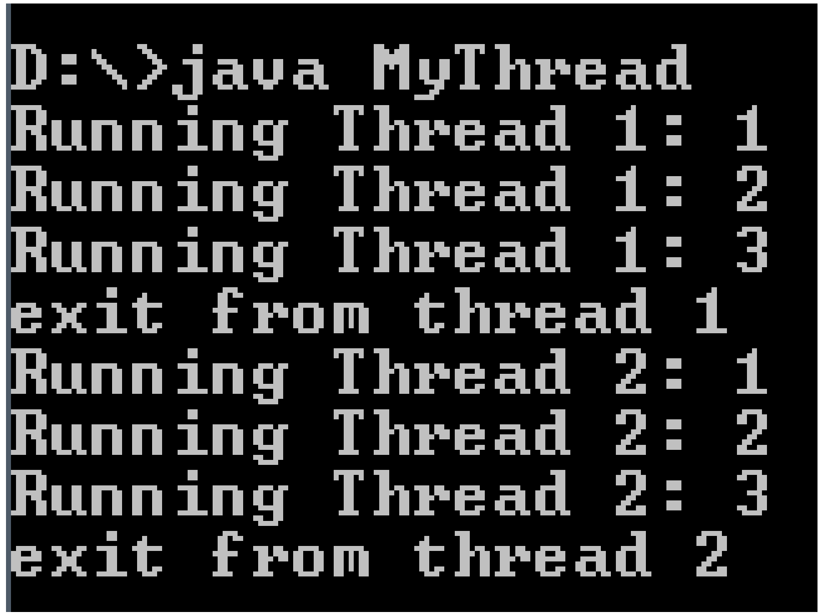
```
class MultiT1 extends Thread{
public void run(){
for(int i=1;i<=3;i++)
{  System.out.println("Running Thread 1: "+i);  }
System.out.println("exit from thread 1");
}
}
class MultiT2 extends Thread{
public void run(){
for(int i=1;i<=3;i++)
{  System.out.println("Running Thread 2: "+i);}
System.out.println("exit from thread 2");
```

contd…

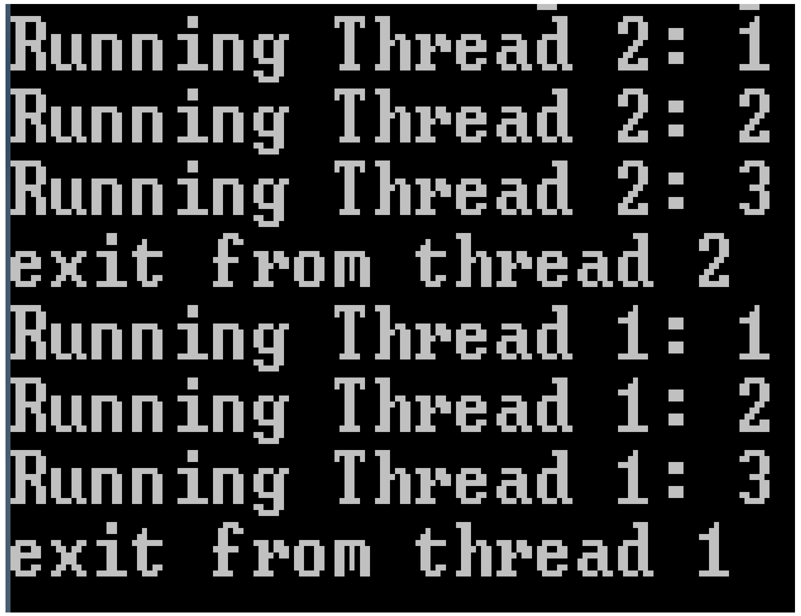
contd…

```
}
}
class MyThread
{
public static void main(String args[]){
MultiT1 t1=new MultiT1();
t1.start();
MultiT2 t2=new MultiT2();
t2.start();
 }
}
```

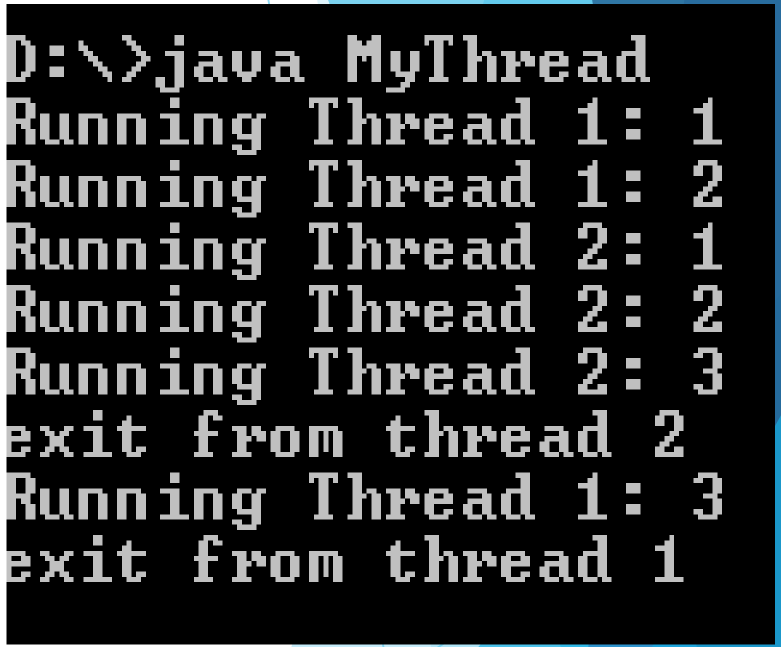
Output1 Output2 output3



```
D:\>java MyThread
Running Thread 1: 1
Running Thread 1: 2
Running Thread 1: 3
exit from thread 1
Running Thread 2: 1
Running Thread 2: 2
Running Thread 2: 3
exit from thread 2
```



```
Running Thread 2: 1
Running Thread 2: 2
Running Thread 2: 3
exit from thread 2
Running Thread 1: 1
Running Thread 1: 2
Running Thread 1: 3
exit from thread 1
```



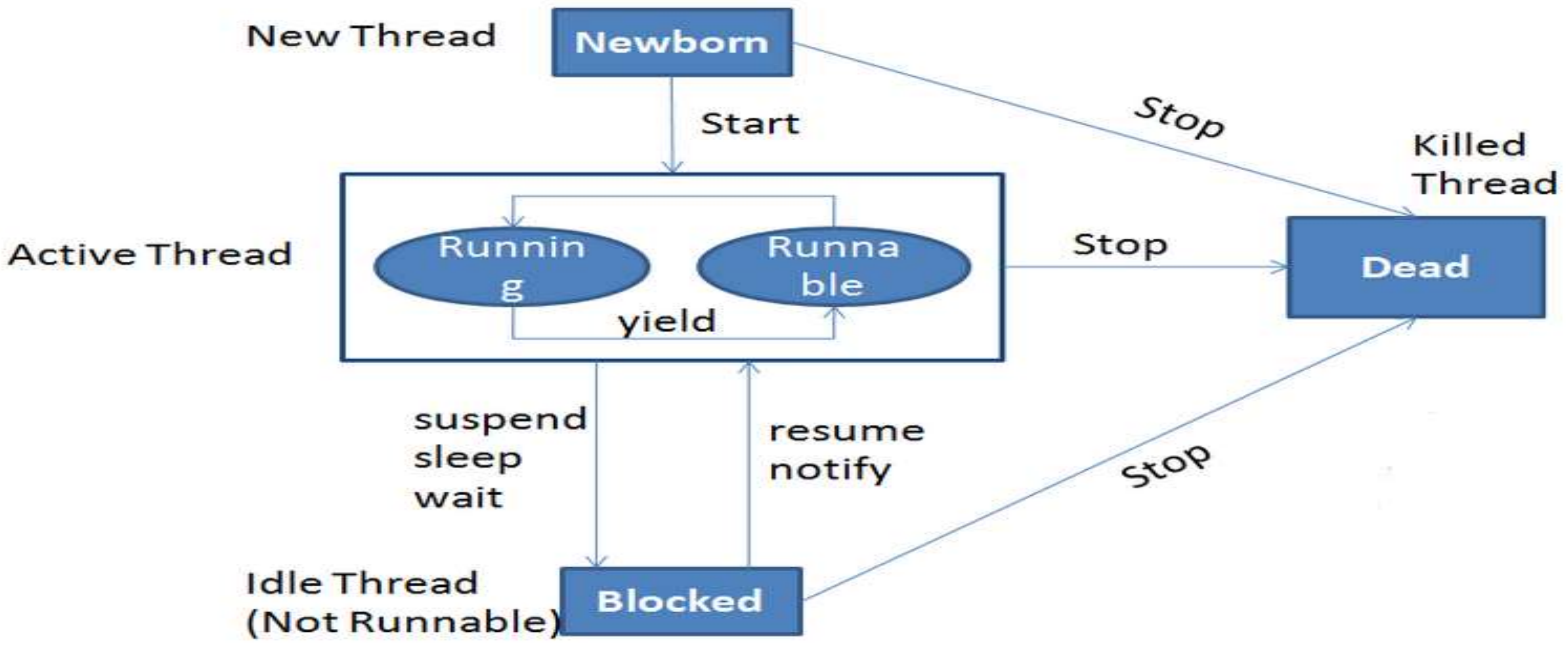
```
D:\>java MyThread
Running Thread 1: 1
Running Thread 1: 2
Running Thread 2: 1
Running Thread 2: 2
Running Thread 2: 3
exit from thread 2
Running Thread 1: 3
exit from thread 1
```

Life cycle of a Thread

During life cycle of a thread, there are many states it can enter.

1. Newborn state
2. Runnable state
3. Running state
4. Blocked state
5. Dead state

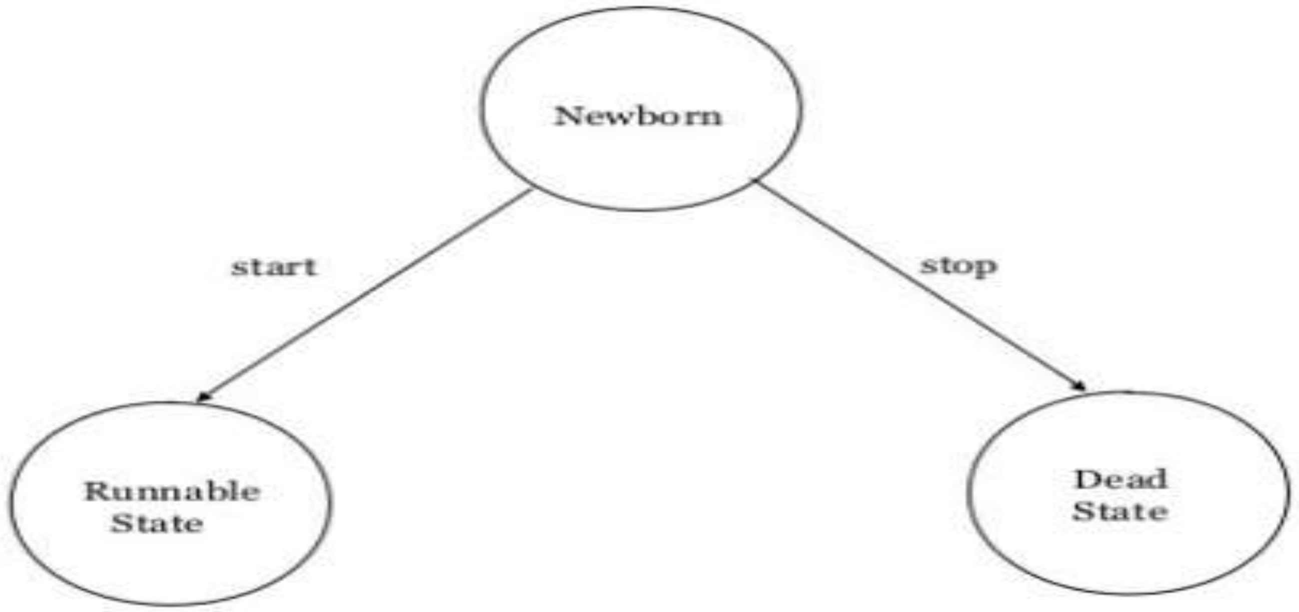
Thread Life cycle



State transition diagram of a thread

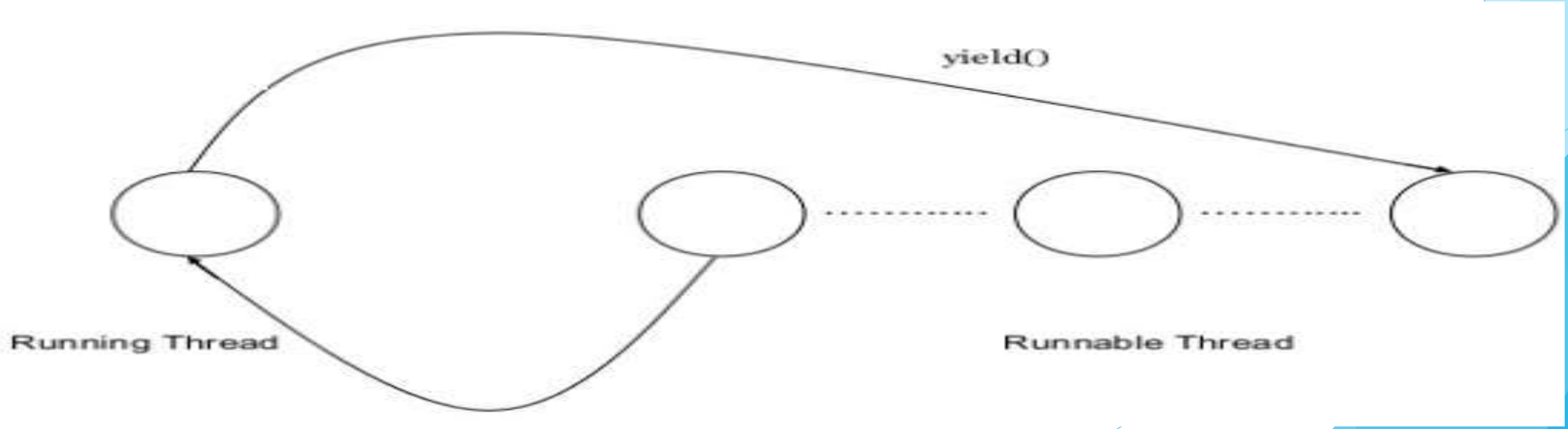
Newborn state

- When we create a thread object, the thread is born and is said to be in newborn state.
- The thread is not yet schedule for running.
- We can do only one of the following things with it:
  - Schedule it for running using *start()* method
  - Kill it using *stop()* method
- If we attempt to use any other method at this stage, an exception will be thrown.



Runnable state

- The runnable state means that the thread is ready for execution and is waiting for the availability of the processor.
- The threads has joined the queue of threads that are waiting for execution.
- If we want a thread to release its control for another thread of equal priority before its turn comes, we can do so by using *yield()* method.

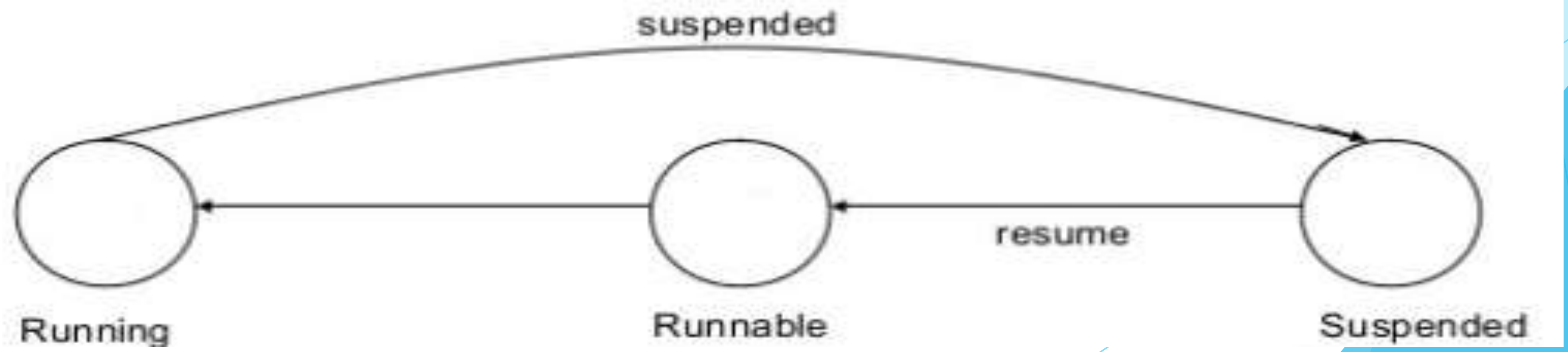


Running state

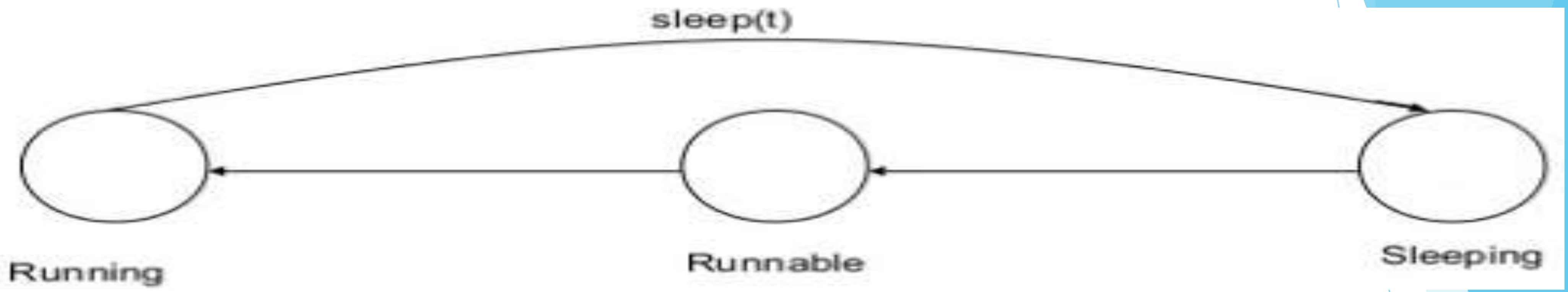
- Running means that the processor has given its time to the thread for its execution.
- The thread runs until it releases control on its own or preempted by a higher priority thread.

Situations when Thread may release its control:

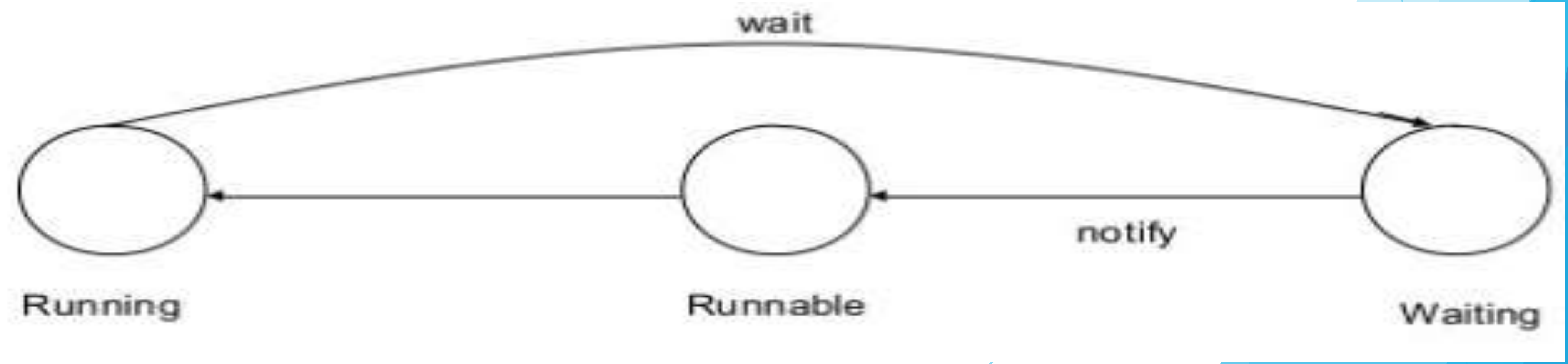
1. It has been suspended using *suspend()* method. A suspended thread can be resumed by using the *resume()* method.



2. It has been made to sleep. We can put a thread to sleep for a specified time period using the method *sleep(time)* where time is in milliseconds.



3. It has been told to wait until some event occurs. This is done using *wait()* method. The thread can be scheduled to run again using the *notify()* method.



Blocked state

- A thread is said to be blocked when it is prevented from entering into the runnable state and subsequently in running state.
- A blocked thread is considered not runnable but not dead.
- A thread can blocked using following methods:
  - sleep() : blocked for specific time
  - suspend() : blocked until further orders
  - wait() : blocked until certain condition occurs

Dead state

- A running thread ends its life when it has completed executing its *run()* method. It is a natural death.
- However, we can kill it by sending the stop message to it using *stop()* method. It is called premature death.
- A thread can be killed as soon as it is born, or while it is running, or even when it is in blocked condition.

Example

```
class A extends Thread
{
public void run(){
for(int i=1;i<=5;i++)
{
  if(i==2)
  yield();
  System.out.println("Running Thread A: "+i);
}
System.out.println("exit from thread A");
}
}
```

Contd...

Contd…

```
class B extends Thread
{
public void run()
{
for(int i=1;i<=3;i++)
{
  System.out.println("Running Thread B: "+i);
  if(i==3)
  stop();
}
System.out.println("exit from thread B");
}
}
```

Contd…

Contd..

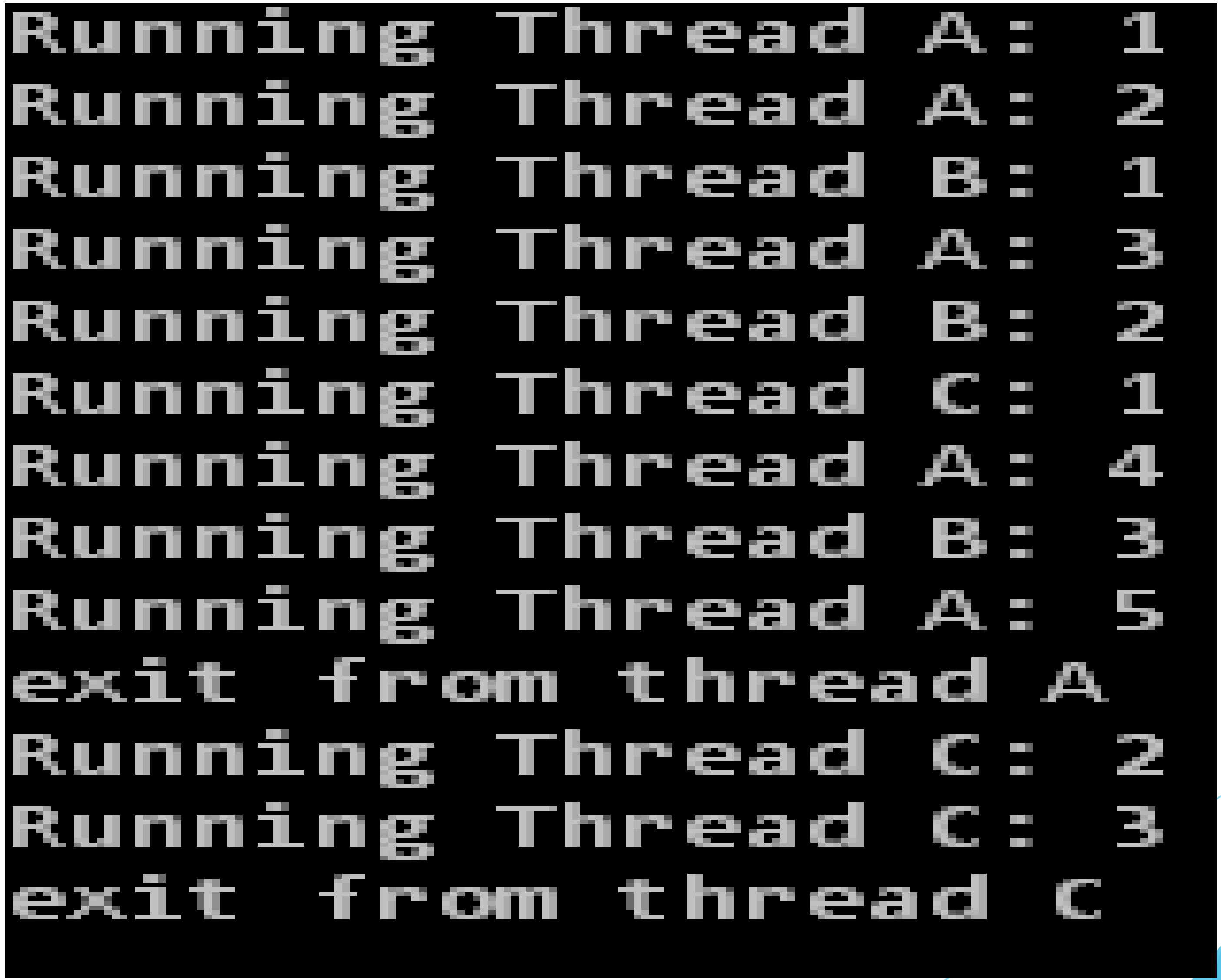
```
class C extends Thread{
public void run(){
for(int i=1;i<=3;i++)
{
    System.out.println("Running Thread C: "+i);
    if(i==1)
    try
    {    sleep(1000);  }
    catch(Exception e)
    {    System.out.println(e);  }
}
System.out.println("exit from thread C");
}}
```

Contd..

Contd..

```
class ExpThread
{
public static void main(String args[]){
A t1=new A();
t1.start();
B t2=new B();
t2.start();
C t3=new C();
t3.start();
 }
}
```

Output



```
Running Thread A: 1
Running Thread A: 2
Running Thread B: 1
Running Thread A: 3
Running Thread B: 2
Running Thread C: 1
Running Thread A: 4
Running Thread B: 3
Running Thread A: 5
exit from thread A
Running Thread C: 2
Running Thread C: 3
exit from thread C
```

Thread Exception

- Java run system will throw *IllegalThreadStateException* whenever we attempt to invoke a method that a thread cannot handle in the given state.
- e.g. a sleeping thread cannot deal with the resume() method because a sleeping thread cannot receive any instructions.

Thread Priority

- In java, each thread is assigned a priority, which affects the order in which it is scheduled for running.
- The threads of the same priority share the processor on a first-come, first-serve basis.
- To set a thread priority setPriority() method is used.
- Syntax:

```
ThreadName.setPriority(intnumber);
```

Where, intnumber is an integer value to which the Thread’s priority is set.

- Priority Constants:

MIN_PRIORITY=1

NORM_PRIORITY=5

MAX_PRIORITY=10

Example

```
class A extends Thread {
public void run(){
for(int i=1;i<=5;i++)
{   System.out.println("Running Thread A: "+i);     }
System.out.println("exit from thread A");
}
}
class B extends Thread {
public void run(){
for(int i=1;i<=3;i++)
{   System.out.println("Running Thread B: "+i);     }
System.out.println("exit from thread B");
}}
```

Contd...

Contd...

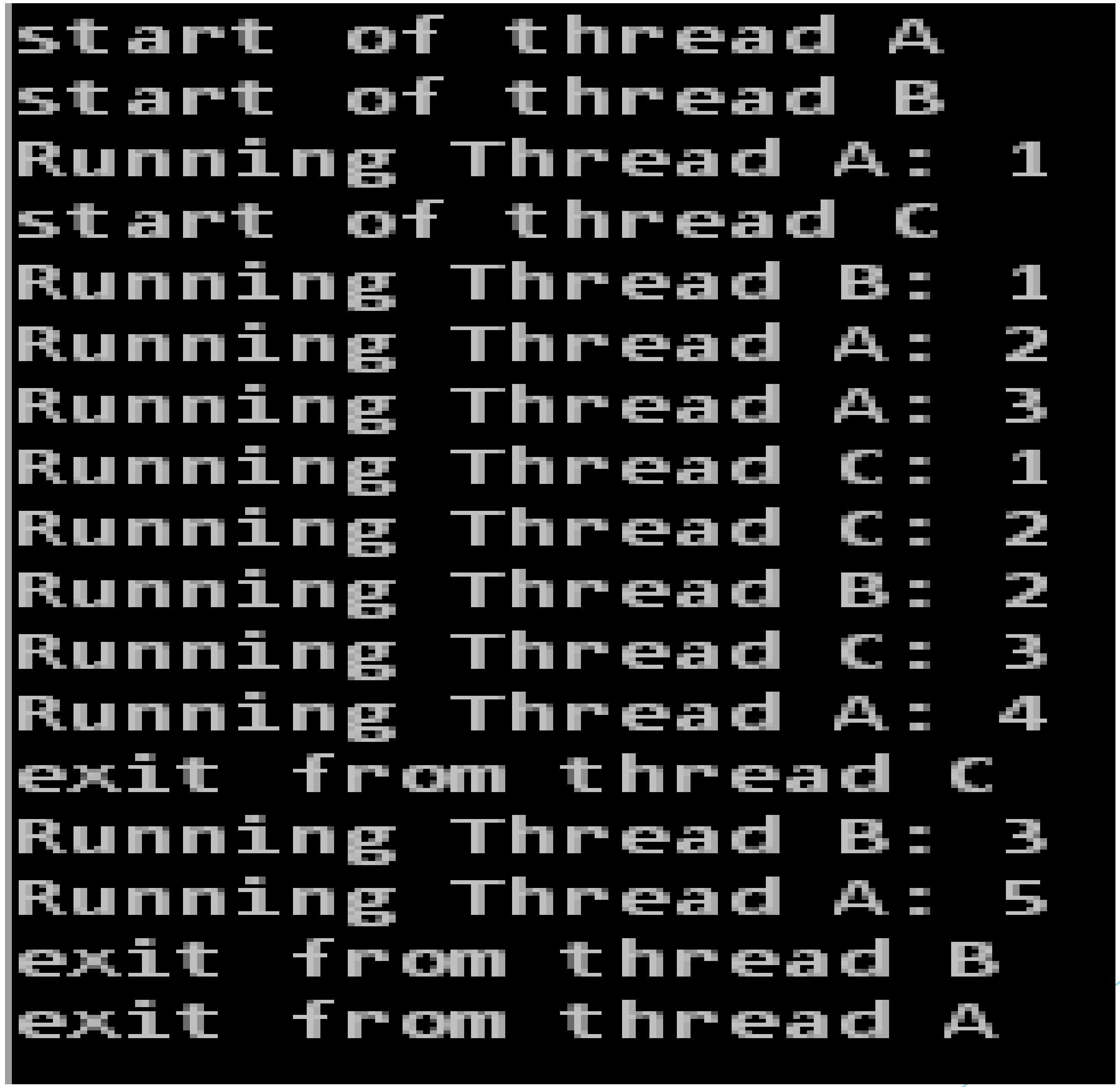
```
class C extends Thread{
public void run(){
for(int i=1;i<=3;i++)
{  System.out.println("Running Thread C: "+i);      }
System.out.println("exit from thread C");
}
}
class ThreadPriority
{
public static void main(String args[]){
A t1=new A();
B t2=new B();
C t3=new C();
```

Contd...

Contd…

```
t3.setPriority(Thread.MAX_PRIORITY);
t2.setPriority(t1.getPriority()+1);
t1.setPriority(Thread.MIN_PRIORITY);
System.out.println("start of thread A");
t1.start();
System.out.println("start of thread B");
t2.start();
System.out.println("start of thread C");
t3.start();
 }
}
```

Output



```
start of thread A
start of thread B
Running Thread A: 1
start of thread C
Running Thread B: 1
Running Thread A: 2
Running Thread A: 3
Running Thread C: 1
Running Thread C: 2
Running Thread B: 2
Running Thread C: 3
Running Thread A: 4
exit from thread C
Running Thread B: 3
Running Thread A: 5
exit from thread B
exit from thread A
```

Synchronization

- Synchronization in java is the capability to control the access of multiple threads to any shared resource.

- Java Synchronization is better option where we want to allow only one thread to access the shared resource.

- E.g. one thread may try to read a record from a file while another is still writing to the same file. At this time we may get strange result. To overcome this problem *synchronization* is used.

Synchronized method

- Synchronized method is used to lock an object for any shared resource.
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.
- When we declare a method synchronized, java created a monitor and hands it over to the thread that calls the method first time. As long as the thread holds monitor, no other thread can enter the synchronized section of code. A monitor is a key and the thread that holds the key can only open the lock.
- E.g.

```
synchronized void update()
{    ……..
}
```

Synchronized block

- Synchronized block can be used to perform synchronization on any specific resource of the method.
- Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.
- If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.
- Syntax:

```
 synchronized (object reference expression)
{
  //code block
 }
```

Implementing the Runnable interface

Steps to implement Runnable interface:

1. Declare the class as implementing the runnable interface.
2. Implement the run() method.
3. Create a thread by defining an object that is instantiated from this runnable class.
4. Call the Thread’s start() method to run the thread.

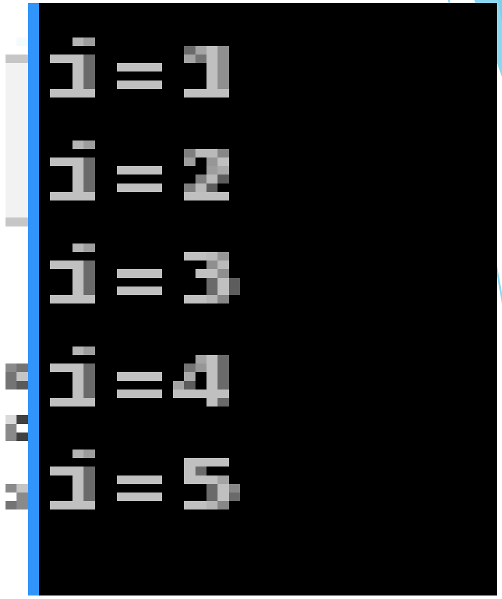
Example

```
class A implements Runnable          //step 1
{
  public void run()                  //step 2
  {
    for(int i=1;i<=5;i++)
    {
       System.out.println("i="+i);
     }
   }
}
```

contd…

contd…

```
class RunnableTest
{
  public static void main(String args[])
  {
     A t=new A();
     Thread ta=new Thread(t);            //step 3
     ta.start();                         //step 4
//new Thread(new A()).start();      // shortcut for above 3 lines
   }
}
```



```
i=1
i=2
i=3
i=4
i=5
```