MULUND

VASHI

BORIVALI DOMBIVALI DADAR

Phone Nos. 02264205700, 02264402060, 9223376244, 9821882868

O C J P

by

Sandeep Gupta

Concepts of Threads –

1) 2 threads associated with same objects of same class in a synchronized block, switching will not take place between these 2 threads

2) 2 threads associated with 2 diff objects of same class having their respective method run()/syn block. Here switching can take place between these 2 syn blocks/run() as both 2 threads are having their respective syn blocks.

File class

A) Introduction

1. The File class is "an abstract representation" of file and directory pathnames.
2. The File class isn't used to actually read or write data.
3. It's used to work at a higher level, making new empty files, searching for files, deleting files, making directories, and working with paths.
4. Objects of type File are used to represent the actual files or directories that exist on a computer's physical disk. They do not represent the data in the files.

Example:

```
import java.io.* ;
class  CreateFile1
{
public static void main(String args[])
{
      try
      {
            boolean  b ;
            File file = new File("eg1.txt");    // 1
            System.out.println(file.exists());
            b = file.createNewFile();
            System.out.println(b);
            System.out.println(file.exists());
      }
      catch(IOException  e)
      { }
}
}
```

(Constructor call)

(During 2nd execution, it doesn't create a new file)

This program produces the output:

false
true
true

and also produces an empty file in your current directory.
If you run the code a second time, you get the output

true
false
true

Note:

1) **First execution** - The first call to exists() returned false since statement 1 does not create a new file on the disk! It simply creates a filename. The createNewFile () method created an actual file and returned true, indicating that a new file was created and that one didn't already exist. Finally, we called exists() again and this time it returned true indicating that the file existed on the disk.

2) **Second execution** - The first call to exists() returns true because we built the file during the first run. Then the call to createNewFile() returns false since the method didn't create a file this time through. Of course, the last call to exists() returns true.

3) We wrote try catch since createNewFile() throws IOException which is a checked exception and must be compulsorily handled.

4) Methods exists() & createNewFile()

boolean exists() - This method returns true if it can find the actual file.

boolean createNewFile() - This method creates a new file if it doesn't already exist.

B) Renaming file

Example 1:

```
import java.io.* ;

class  RenameFile1
{
public static void main(String args[])
{
      try
      {
            File file1 = new File("Jack.txt");
            file1.createNewFile();
            System.out.println(file1.exists());

            File file2 = new File("Jill.txt");
            file2.createNewFile();
            System.out.println(file2.exists());

            File file3 = new File("Tom.txt");
            System.out.println(file3.exists());
            System.out.println(file1.renameTo(file3));
            System.out.println(file3.exists());
            System.out.println(file1.renameTo(file3));

            System.out.println(file3.renameTo(file2));  // 1
      }
      catch(IOException  e)
      {}
}
}
```

true ← false

true → renames and also returns true/false

true → false

true → false

true

At the end of program, there will only 1 file Jill.txt

What happens if statement 1 is not in the program?

Ans: There will be 2 files in directory: Tom.txt and Jill.txt

Example 2:

```
import java.io.* ;

class  RenameFile2
{
public static void main(String args[])
{
        File file1 = new File("Jack.txt");
        File file2 = new File("Jill.txt");

        System.out.println(file1.exists());
        System.out.println(file2.exists());
        System.out.println(file1.renameTo(file2)); // Statement 1
}
}
```

false false false

Note:

1. Statement 1 returns false since "Jack.txt" does not exist.
2. If "Jack.txt" exists but "Jill.txt" does not exist, still renaming will be done and statement 1 returns true.
3. If "Jack.txt" does not exist but "Jill.txt" exists, renaming will not be done and statement 1 returns false.
4. In either case, no exception is thrown. Moreover, for renaming to succeed the source file must exist. Destination may not exist.
5. Note that there is no try catch since method createFile() is not used.

(Bcoz, only the name of destination file is needed)

C) Deleting File

```
import java.io.* ;

class  DeleteFile1
{
public static void main(String args[])
{
      try
      {
            File file1 = new File("Ram.txt");
            file1.createNewFile();

            File file2 = new File("Rahim.txt");

            System.out.println(file1.delete());
            System.out.println(file2.delete());
      }
      catch(IOException  e) { }
}
}
```

true false

Note:
The methods createNewFile(), renameTo(), delete() may or may not be written using a receiving variable [b = file1.createNewFile() of first program]

i.e., these function() return true or false, but they can be written without using receiving variable

D) Working with Directories

Example 1: Creating directory

```
import java.io.* ;

class  CreateDir
{
public static void main(String args[])
{
      try
      {
            File dir1 = new File("OCJP");
            dir1.mkdir();
            System.out.println(dir1.isDirectory());
            File file1 = new File(dir1 , "student.txt");
            file1.createNewFile();

            File dir2 = new File("OOPM");
            File file2 = new File(dir2 , "teacher.txt");
            System.out.println("Test Statement 1");
            file2.createNewFile();
            System.out.println("Test Statement 2");
      }
      catch(IOException  e)
      { System.out.println("IOException caught here!!"); }
}
}
```

Note:
The statement "file2.createNewFile();" throws IOException since the directory "OOPM" is not yet created

Example 2: Creating sub-directory

```
import java.io.* ;
class  CreateSubDir
{
public static void main(String args[])
{
      try
      {
            File dir1 = new File("Oracle");
            dir1.mkdir();

            File dir2 = new File(dir1 , "Java");
            dir2.mkdir();

            File file1 = new File(dir2 , "student.txt");
            file1.createNewFile();
      }
      catch(IOException  e)     { }
} }
```

→ Only Empty directories can be deleted

Example 3: Deleting and Renaming a directory

[A non empty directory can be renamed, but cannot be deleted]

```
import java.io.* ;

class  DeleteDir
{
public static void main(String args[])
{
        try
        {
                File dir1 = new File("ABC");
                dir1.mkdir();

                File file1 = new File(dir1 , "xyz.txt");
                file1.createNewFile();

                System.out.println(dir1.delete());

                File dir2 = new File("CBA");
                System.out.println(dir1.renameTo(dir2));
        }
        catch(IOException  e)
        { }
}
}
```

~~ABC~~ CBA

xyz.txt

→ false

→ true

Note:

1. You cannot delete a directory which is not empty.
2. You can rename a directory which is not empty.

FileReader & FileWriter

FileReader

1. This class is used to read character files.
2. Its read() methods are fairly low-level, allowing you to read single characters, the whole stream of characters, or a fixed number of characters.
3. FileReaders are usually wrapped by higher level objects such as BufferedReaders, which improve performance and provide more convenient ways to work with the data.

FileWriter

1. This class is used to write to character files.
2. Its write() methods allow you to write character(s) or strings to a file.
3. FileWriters are usually wrapped by higher-level Writer objects, such as BufferedWriters or PrintWriters, which provide better performance and higher-level, more flexible methods to write data.

Example 1: read() & write() reading and writing in ONE attempt

```
import java.io.* ;
class  FileReaderWriter1  {
public static void main(String args[])
{
        char in[]=new char[50];
        int size = 0 ;
        try
        {
                File file = new File("Example1.txt");
                FileWriter fw = new FileWriter(file);
                // Creates the FileWriter object as well as the file Example1.txt

                fw.write("Welcome\nto\nJava");   // Writes characters on the file
                fw.flush();                      // optional, but better to write
                fw.close();

                FileReader fr = new FileReader(file); // Creates the FileReader object

                size = fr.read(in); // Reads all the characters and stores in array in.
                                    // Also returns the number of characters read.
                System.out.println(size);

                for (char c : in)
                  System.out.print(c);
                // fr.flush(); ERROR - cant use with FileReader
                fr.close();
        }
        catch(IOException  e)     { }
}}
```

Note:

1. Methods of FileReader and FileWriter may throw IOException.
2. flush() is optional.

Example 2: read() & write() reading and writing character by character

```
import java.io.* ;

class FileReaderWriter2
{
public static void main(String args[])
{
      String str="Hello\nWorld";
      int c ;
      try
      {
            File file = new File("Example2.txt");

            FileWriter fw = new FileWriter(file);

            for(int i=0;i<str.length();i++)
              fw.write(str.charAt(i));    // Writes a single character on the file

            fw.flush();

            fw.close();

            FileReader fr = new FileReader(file);

            while ( (c=fr.read()) != -1 )   // reads a single character
              System.out.print((char)c);

            fr.close();
      }
      catch(IOException e)
      { }
}
}
```

[creates obj as well as file "Example2.txt"]

[Predefined methods of class String]

[write() of prev prog take String, this one takes char]

[ascii value is of data type int]

[reads ASCII value of a single character at a time]

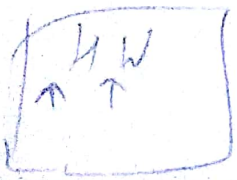
[Typecasting 'ASCII' to 'Char']

[At end of file method read() automatically returns value -1]

[Previous prog read() takes array as parameter, but this read() takes single char at a time and does reading work]

(Eg) :-

Example 2.txt



→ While reading Example 2.txt, fr.read() takes ASCII value of all characters one by one into variable c and the SOP(char(c)) converts ASCII value to again to char and displays it. Here data type of c is int and in SOP() c is type casted to char

Chaining

Java's entire I/O system was designed around the idea of using several classes in combination. Combining I/O classes is sometimes called wrapping and sometimes called chaining.

Apart from the classes dealt above, the following classes are used in chaining.

BufferedReader

1. This class is used to make lower-level Reader classes like FileReader more efficient and easier to use.
2. Compared to FileReaders, BufferedReaders read relatively large chunks of data from a file at once and keep this data in a buffer. When you ask for the next character or line of data, it is retrieved from the buffer, which minimizes the number of times that time-intensive, file-read operations are performed.
3. In addition, BufferedReader provides more convenient methods, such as readLine(), that allow you to get the next line of characters from a file.

BufferedWriter

1. This class is used to make lower-level classes like FileWriters more efficient and easier to use.
2. Compared to FileWriters, BufferedWriters write relatively large chunks of data to a file at once, minimizing the number of times that slow, file-writing operations are performed.
3. The BufferedWriter class also provides a newLine() method to create platform-specific line separators automatically.

PrintWriter

1. This class has been enhanced significantly in java 5.
2. Because of newly created methods and constructors (like building a PrintWriter with a File or a String), you might find that you can use PrintWriter in places where you previously needed a FileWriter and/or a BufferedWritter.
3. New methods like format (), print (), and append () make PrintWriters very flexible and powerful.

The 3 programs given below demonstrate increasing level of chaining.

Example 1:

```
import java.io.* ;

class  Chaining1
{
public static void main(String args[])
{
        String  s;
        try
        {
            File file = new File("Example3.txt");
            FileWriter fw = new FileWriter(file);
            BufferedWriter out = new BufferedWriter(fw);
            PrintWriter pw = new PrintWriter(out);
            // The 4 statements above demonstrate chaining or wrapping

            pw.println("Welcome");
            pw.write("to");
            pw.println("Java");

            pw.flush();
            pw.close();

            FileReader fr = new FileReader(file);
            BufferedReader in = new BufferedReader(fr);
            while ((s=in.readLine())!=null)    // reads a single string
              System.out.println(s);
            in.close();
        }
        catch(IOException  e)
        { }
}
}
```

Example 2:

```
import java.io.* ;

class  Chaining2
{
public static void main(String args[])
{
        String  s;
        try
        {
            File file = new File("Example4.txt");
            FileWriter fw = new FileWriter(file);
            PrintWriter pw = new PrintWriter(fw);

            pw.println("Hello");
            pw.println("World");
            pw.flush();
            pw.close();
```

```
            FileReader fr = new FileReader(file);
            BufferedReader in = new BufferedReader(fr);
            while ((s=in.readLine())!=null)
              System.out.println(s);
            in.close();
        }
        catch(IOException  e)
        { }
    }
    }
```

Example 3:

```
import java.io.* ;

class  Chaining3
{
public static void main(String args[])
{
      String  s;
      try
      {
            PrintWriter pw = new PrintWriter("Example5.txt");
            pw.println("Hello");
            pw.println("World");
            pw.flush();
            pw.close();

            FileReader fr = new FileReader("Example5.txt");
            BufferedReader in = new BufferedReader(fr);
            while ((s=in.readLine())!=null)
              System.out.println(s);
            in.close();
      }
      catch(IOException  e)
      { }
}
}
```

Note:

1)
PrintWriter can be directly given filename. That's why the following was valid:
PrintWriter pw = new PrintWriter("Example5.txt");

2)
Similarly FileReader can also be directly given filename. That's why the following was valid:
FileReader fr = new FileReader("Example5.txt");

3)
However something like
BufferedReader in = new BufferedReader("Example5.txt");
would be invalid since BufferedReader and BufferedWriter needs a File object.

The following table summarizes the Classes used above:

java.io class	extends From	Key Constructor(s) arguments	Key Methods
File	Object	File , String String String , String	createNewFile() delete() exists() isDirectory() isFile() list() mkdir() renameTo()
FileWriter	Writer	File String	close() flush() write()
Bufferedwriter	Writer	Writer	close() flush() newLine() write()
PrintWriter	Writer	File (as of java 5) string (as of java 5) OutputStream writer	close() flush() format() println() write()
FileReader	Reader	File String	read()
BufferedReader	Reader	Reader	read() readLine()

Console class

1. This new Java 6 convenience class provides methods to read input from the console and write formatted output to the console.
2. The console class makes it easy too accept input from the command line, both echoed and non-echoed (such as a password), and makes it easy to write formatted output to the command line.
3. On the input side, the methods you'll have to understand are readLine() and readPassword(). The readLine() method returns a string containing whatever the user keyed.
4. However, the readpassword() method doesn't return a string; it returns a character array. The reason is: Once you've got the password, you can verify it and then absolutely remove it from memory. If a string was returned, it could exist in a pool somewhere in memory, and perhaps some nefarious hacker could find it.

```
File file = new File("Java", "myfile.txt");
File.createNewFile();
```

Example 1:

```
import java.io.* ;
class  ConsoleTest1
{
public static void main(String args[ ])
{
      Console c = System.console();

      String s1;
      char p[];

      System.out.print("User ID: ");
      s1=c.readLine();                    // Input is echoed

      System.out.print("Password: ");
      p=c.readPassword();
      // Input not echoed. readPassword() returns a character array.

      c.format("User ID is %s",s1);

      c.format("\nPassword is ");
      for(char  x : p)
        c.format("%c" , x);
} }
```

Note:
There are 3 important methods in Console:
1) readLine() 2) readPassword() 3) format()

readLine() and readPassword() can be given in two ways. The second way is shown in next program.

format() is EXACTLY same as printf() and is already covered in chapter 11 - Data Formatting

Example 2:

```
import java.io.Console ;
class  ConsoleTest2  {
public static void main(String args[ ])
{
      Console c = System.console();

      String  s1;
      char  p[];

      s1=c.readLine("%s","User ID: ");

      p=c.readPassword("%s","Password: ");

      c.format("User ID is %s",s1);

      c.format("\nPassword is ");
      for(char  x : p)
        c.format("%c" , x);
} }
```

Multitasking

1. Multitasking is a phenomenon in which a single processor executes multiple tasks at the same time.
2. In multitasking, the processor executes one task for a fraction of a second and then ***switches*** to another task which it executes for another fraction of second. This switching is so fast that it appears to us that all the tasks are being executed simultaneously.
3. Multitasking is of two types: Process based Multitasking and Thread Based Multitasking.
4. In process based multitasking, the computer seems to work on different processes at the same time. A process is basically a program in execution. It is process based multitasking which allows us to listen to music (lets say using winamp) and edit text (lets say using microsoft word) at the same time.
5. In thread based multitasking(also called multithreading) the computer seems to work on different threads at the same time. ***A thread is basically a part of a program which follows separate path of execution.*** It is thread based multitasking which allows us to print one part of text file(managed by one thread) while it is still being edited(managed by another).
6. Every Java program has at least one thread called "main thread".
7. Threads are useful in server-side programming in which the server considers each client as one thread. This arrangement allows the server to serve multiple clients simultaneously.

Creating Threads

A thread is basically a part of a program which follows ***separate*** path of execution.

A thread can be created using either of the following techniques:

a) By **extending** the java.lang.**Thread** Class

b) By **implementing** the java.lang.**Runnable** interface

→ The object of a class which is subclass of class Thread is considered as a thread

A) extending the java.lang.Thread Class

Step 1: Create a class (Say MyThread) which extends the Thread class of java.lang

Step 2: Once the class is created it needs to have an overridden method run()

Step 3: Create a Thread object **OR**

Create an object of MyThread and pass it as a parameter to Thread constructor.

Step 4: Call the method start() using object of class Thread or MyThread.

Example 1:

```
class MyThread extends Thread
{
public void run()
{
        String  s = Thread.currentThread().getName();

        for(int i=1;i<=10;i++)
                System.out.println(s);

        System.out.println(s+"  terminated");
}
}

class ExThread1
{
public static void main(String args[])
{
MyThread t = new MyThread();

t.setName("Siri");
t.start();
for(int i=1;i<=10;i++)
        System.out.println("Main");

System.out.println("Main Terminated");
}
}
```

start() and setName() are in class Thread

o/p: Main Siri Siri Siri Siri Main Main Siri Siri main main main Siri Siri Siri Siri main main main Siri terminated Main main Main terminated

any other o/p [i.e o/p is not same]

In the above program, there are 2 threads - main and 't'. The 't' invokes start() which in turn invokes run(). So run is also thread. In fact any method invoked by 't' is a part of Thread.

Example 2:

```
class MyThread extends Thread
{
public void run()
{
        String  s = Thread.currentThread().getName();

        for(int i=1;i<=10;i++)
                System.out.println(s);

        System.out.println(s+"  terminated");
}
}

class ExThread2
{
public static void main(String args[])
{
MyThread t = new MyThread();
Thread t1 = new Thread(t);

t1.setName("Siri");
t1.start();

for(int i=1;i<=10;i++)
        System.out.println("Main");

System.out.println("Main Terminated");
}
}
```

[ie here t1 can use methods of class MyThread]

[t1 and t are now same] ← [association of t1 and t]

[t1.start() → run() of Thread is called in ∅ class Thread]

[Bcoz, t1 is a ~~class~~ obj of class Thread, so any method t1 invokes directly or indirectly is a thread. Also, in every prog there is Thread main()]

B) implementing the java.lang.**Runnable** interface

Step 1: Create a class (Say MyRunnable) which implements interface Runnable of java.lang

Step 2: Once the class is created it needs to have an overridden method run()

Step 3: Create an object of MyRunnable and pass it as a parameter to Thread constructor.

Step 4: Call the method start() using object of class Thread.

Example 1:

```
class MyRunnable implements Runnable
{
public void run()
{
        String  s = Thread.currentThread().getName();

        for(int i=1;i<=10;i++)
                System.out.println(s);

        System.out.println(s+"  terminated");
}
}

class ImpRun1
{
public static void main(String args[])
{
MyRunnable r = new MyRunnable();
Thread t1 = new Thread(r);
t1.setName("Siri");
t1.start();

for(int i=1;i<=10;i++)
        System.out.println("Main");

System.out.println("Main Terminated");
}
}
```

Example 2: Using Thread constructor which takes 2 parameters & calling start() twice

```
class MyRunnable implements Runnable
{
public void run()
{
        String  s = Thread.currentThread().getName();

        for(int i=1;i<=10;i++)
                System.out.println(s);

        System.out.println(s+"  terminated");
}}

class ImpRun2
{
public static void main(String args[])
{
        MyRunnable r = new MyRunnable();
        Thread t1 = new Thread(r , "Siri");
        // t1.setName("Siri");

        t1.start();

        for(int i=1;i<=10;i++)
                System.out.println("Main");

        t1.start();

        System.out.println("Main Terminated");
}}
```

To give name to Thread

O/P

Some O/P is displayed and then the 2nd call to start throws IllegalThreadStateException.

start() executes only once for one thread

Note:

1. Instead of using setName(), we may use another constructor of class Thread as shown above.
2. If we write t1.start() more than once then compilation is successful, but we get java.lang.IllegalThreadStateException.
3. Once a thread has been started, it can never be started again.
4. If you have a reference to a Thread and you call start(), it's started. If you call start() a second time, it will cause an IllegalThreadStateException, which is a kind of RuntimeException.
5. This happens whether or not the run() method has completed from the first start() call. Only a new thread can be started and that too only once. A runnable thread or a dead thread cannot be restarted.

Which thread creation technique should we use?

1. Extending the Thread class is the easiest, but it's usually not a good OO practice.
2. This is because sub-classing should be reserved for specialized versions of more general superclasses.
3. So the only time it really makes sense (from an OO perspective) to extend Thread class is when you have a more specialized version of Thread class.
4. Chances are, though, that the thread work you really want is just a job to be done by a thread.
5. In that case, you should design a class that implements the Runnable interface, which also leaves your class free to extend some other class.

Example 3: Creating Multiple Threads

```
class MyRunnable implements Runnable
{
public void run()
{
        String  s = Thread.currentThread().getName();

        for(int i=1;i<=10;i++)
                System.out.println(s);

        System.out.println(s+"  terminated");
}
}

class MulThread1
{
public static void main(String args[])
{
MyRunnable r = new MyRunnable();

Thread t1 = new Thread(r);
Thread t2 = new Thread(r);

t1.setName("Gini");
t2.setName("Johny");

t1.start();
t2.start();

for(int i=1;i<=10;i++)
        System.out.println("Main");

System.out.println("Main Terminated");
}
}
```

O/P :-

```
Main
Johny
Johny
Gini
Gini
Gini
J
G
G
G
G
G
G
G
M
M
M
M
J
J
J
M
M
M
M
M
J
J
J
J
J  Terminated
M  Ter...
G  Termin...
```

Note :- In this program, there are total of 3 threads - Main, run() invoked by t1 (Gini) and run() invoked by t2 (Johny). These 3 threads executes simultaneously with switching in between.

Example 4: Usage of some basic methods

```
class MyRunnable implements Runnable
{
public void run()
{
        System.out.println(Thread.currentThread().getName());
}
}

class ImpRun4
{
public static void main(String args[])
{
MyRunnable r = new MyRunnable();

Thread t1 = new Thread(r);
Thread t2 = new Thread(r);

System.out.println(t1.getName());
System.out.println(t2.getName());

System.out.println(Thread.currentThread().getName());

System.out.println(t1.getId());
System.out.println(t2.getId());
System.out.println(t1.getPriority());
System.out.println(t2.getPriority());

System.out.println("---------------");
t1.start();
t2.start();
}
}
```

Output:

```
Thread-0
Thread-1
main
9
10
5
5
---------------
Thread-0
Thread-1
```

Note:

1) Thread.currentThread().getName() and t1.getName() (or t2.getName()) produce same output.
2) 5 is the default priority of each thread.
3) The last two lines of output may appear in reverse order.
4) The getId () method returns a positive, unique, long number, and that number will be that thread's only ID number for the thread's entire life.

Example 5: Calling run() instead of start()

```
class MyRunnable implements Runnable
{
public void run()
{
    String  s = Thread.currentThread().getName();
        for(int i=1;i<=5;i++)
                System.out.println(s);
        System.out.println(s+"  terminates");
}}
class ImpRun3
{
public static void main(String args[])
{
MyRunnable r = new MyRunnable();
Thread t1 = new Thread(r);

t1.setName("Siri");
t1.run();

for(int i=1;i<=5;i++)
        System.out.println("Main");

t1.run();

System.out.println("Main Terminated");
}}
```

Output:

```
main
main
main
main
main
main  terminates
Main
Main
Main
Main
Main
main
main
main
main
main
main  terminates
Main Terminated
```

Note:

1) Watch for difference in case of 'M'
2) The "main" where m is in lower case is default name of main thread. Hence in this program, there is only one thread.
3) run() can be called multiple times.

The Thread Scheduler

1. The thread scheduler is the part of the JVM that decides which thread should run at any given moment, and also takes threads out of the run state.

2. Assuming a single processor machine, only one thread can actually run at a time. And it's the thread scheduler that decides which thread - of all that are eligible - will actually run. When we say eligible, we really mean in the runnable state.

3. Any thread in the runnable state can be chosen by the scheduler to be the one and only running thread. If a thread is not in a runnable state, then it cannot be chosen to be the currently running thread. The order in which runnable threads are chosen to run is not guaranteed.

4. Although queue behavior is typical, it isn't guaranteed. Queue behavior means that when a thread has finished with its "turn" it moves to the end of the line of the runnable pool and waits until it eventually gets to the front of the line, where it can be chosen again.

5. In fact, we call it a runnable pool, rather than a runnable queue, to help reinforce the fact that threads aren't all lined up in some guaranteed order. Although we don't control the thread scheduler (we can't, for example, tell a specific thread to run), we can sometimes influence it.

6. The following methods from the java.lang.Thread class give us some tools for influencing the scheduler.

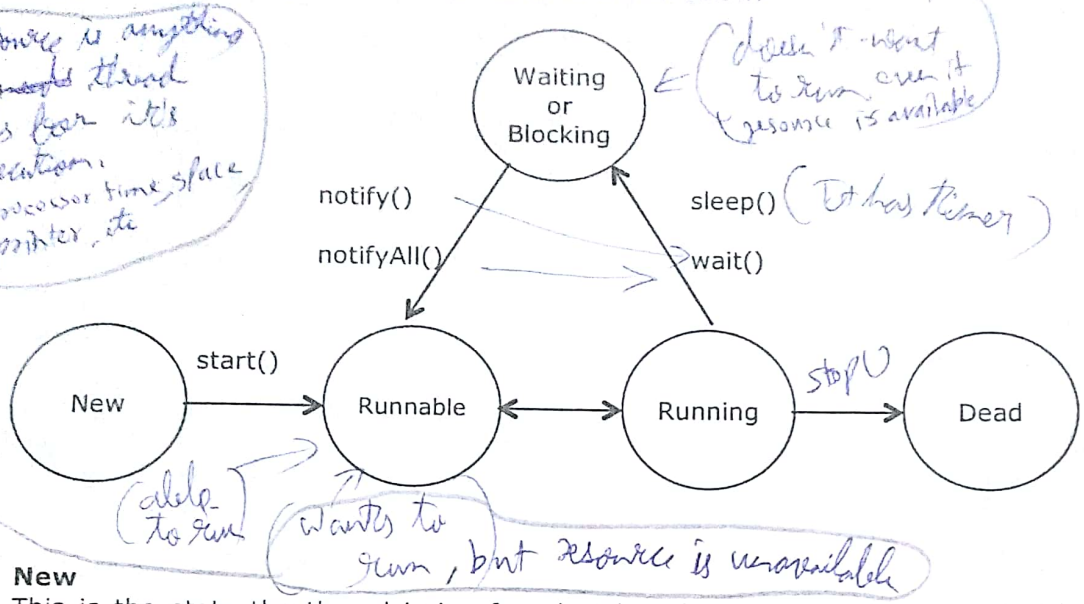
```
public static void sleep(long milliseconds) throws InterruptedException
public static void yield()
public final void join() throws InterruptedException
public final void setpriority(int  newpriority)
```

7. The following methods from the java.lang.Object class also influence the scheduler.

```
public final void wait() throws InterruptedException
public final void notify()
public final void notifyAll()
```

Thread States

A thread can be only in one of five states as shown below:



New
This is the state the thread is in after the Thread instance has been created but the start() method has not been invoked on the thread. It is a live Thread object, but not yet a thread of execution. At this point, the thread is considered not alive.

Runnable
This is the state a thread is in when it's eligible to run but the scheduler has not selected it to be the running thread. A thread first enters the runnable state when the start() method is invoked, but a thread can also return to the runnable state after running or coming back from a blocked/waiting or sleeping state. When the thread is in the runnable state, it is considered alive.

Running
This is the state a thread is in when the thread scheduler selects it from the runnable pool to be the currently executing thread. A thread can transition out of a running state for several reasons, including because "the thread scheduler felt like it." There are several ways to get to the runnable state, but only one way to get to the running state: The scheduler chooses a thread from the runnable pool.

Waiting/blocked/sleeping
This is the state a thread is in when it's not eligible to run. The thread is still alive, but is currently not eligible to run. In other words, it is not runnable, but it might return to a runnable state later if a particular event occurs. A thread may be blocked waiting for a resource (like I/O or an object's lock), in which case the event that sends it back to runnable is the availability of the resource - for example, if data comes in through the input stream the thread code is reading from, or if the object's lock suddenly becomes available.
A thread may be sleeping because the thread's run code tells it to sleep for some period of time, in which case, the event that sends it back to runnable state causes it to wake up because its sleep time has expired. Or the thread may be waiting because the

thread's run code causes another thread to send a notification that it may no longer be necessary for the thread to wait. The important point is that one thread does not tell another thread to block. Some methods may look like they tell another thread to block, but they don't. If you have a reference t to another thread, you can write something like this: t.sleep (); or t.yield();
But those are actually static methods of the Thread class - they don't affect the instance t; instead, they are defined to always affect the thread that's currently executing.
Note also that a thread in a blocked state is still considered alive.

Dead
A thread is considered dead when its run() method completes. It may still be a viable Thread object, but it is no longer a separate thread of execution. Once a thread is dead, it can never be brought back to life!

Preventing Thread Execution

There are 4 methods which are used to prevent thread execution.

a) sleep(): public static void sleep(long milliseconds) throws InterruptedException

The sleep() method is a static method of class Thread. You use it in your code to "slow a thread down" by forcing it to go into a sleep mode before coming back to runnable. When a thread wakes up, it simply goes back to the runnable state. So the time specified in sleep() is the minimum duration in which the thread won't run, but it is not the exact duration in which the thread won't run.

b) yield(): public static void yield()

What yield() is supposed to do is make the currently running thread head back to runnable to allow other threads of the same priority to get their turn. So the intention is to use yield() to promote graceful turn-taking among equal-priority threads. In reality, the yield() method isn't guaranteed to do what it claims. Even if yield() does cause a thread to step out of running and back to runnable, there's no guarantee that the yielding thread won't just be chosen again over all the others! So while yield() might make a running thread give up its slot to another runnable thread of the same priority, there's no guarantee.
A yield() won't ever cause a thread to go to the waiting/sleeping/blocking state. At the most, a yield() will cause a thread to go from running to runnable, but again, it might have no effect at all.

c) join(): public final void join() throws InterruptedException

The non-static join() method of class Thread lets one thread "join onto the end" of another thread. If you have a thread B that can't do its work until another thread A has completed its work, then you want thread B to "join" thread A. This means that thread B will not become runnable until A has finished.

```
Thread t = new Thread() ;
t.start () ;
t.join () ;
```

The preceding code takes the currently running thread (if this were in the main() method, then that would be the main thread) and joins it to the end of the thread referenced by t. This blocks the current thread from becoming runnable until the thread referenced by t is no longer alive.

Example: Usage of sleep(), join() & yield()

```
class MyRunnable1 implements Runnable
{
public void run()
{
      String  s = Thread.currentThread().getName();
      for(int i=1;i<=5;i++)
      {
            if(i==4)
               Thread.yield();
            System.out.println(s + " : " + i);
      }
      System.out.println(s + "  terminated");
}
}

class MyRunnable2 implements Runnable
{
public void run()
{
      String  s = Thread.currentThread().getName();
      for (int k=1; k<=5;k++)
      {
            System.out.println(s + " : " + k);
            try
            {
               Thread.sleep(500);
            }
            catch(InterruptedException e)
            {
            System.out.println(e);
         }
      }
      System.out.println(s + "  terminated");
}
}

class ThreadMethods1
{
public static void main(String args[])
{
MyRunnable1  r1 = new MyRunnable1();
MyRunnable2  r2 = new MyRunnable2();

Thread t1 = new Thread(r1 , "Gini");
Thread t2 = new Thread(r2 , "Johny");

System.out.println("Start Thread Gini");
t1.start();
System.out.println("Is Gini alive : "+t1.isAlive());

System.out.println("Start Thread Johny");
t2.start();
System.out.println("Is Johny alive : "+t2.isAlive());
```

① Concept of Join :-

```
System.out.println("Calling join() ");
try
{
  t1.join();
  t2.join();
}
catch(InterruptedException e)
{
  System.out.println(e);
}
System.out.println("Main Terminated");
}
}
```

A will not start continue until B and C are over

A
{
B.join();
C.join();
}

B

C

Note :- Until t1 and t2 are not finished, main() cannot proceed further

~~Concept of~~ .

Output 1:

```
Start Thread Gini
Is Gini alive : true
Start Thread Johny
Gini : 1
Gini : 2
Gini : 3
Is Johny alive : true
Caliing join()
Gini : 4
Gini : 5
Johny : 1
Gini  terminated
Johny : 2
Johny : 3
Johny : 4
Johny : 5
Johny  terminated
Main Terminated
```

(main thread)

(Gini Thread)

Johny Thread

Pause

Johny thread was sleeping

Gini terminated but it remains of other thread, so it executes

Output 2: Without join

```
Start Thread Gini
Is Gini alive : true
Start Thread Johny
Gini : 1
Gini : 2
Gini : 3
Is Johny alive : true
Caliing join()
Main Terminated
Johny : 1
Gini : 4
Gini : 5
Gini  terminated
Johny : 2
Johny : 3
Johny : 4
Johny : 5
Johny  terminated
```

Johny sleeping here

Output: Without yeild

```
Start Thread Gini
Is Gini alive : true
Gini : 1
Start Thread Johny
Gini : 2
Gini : 3
Gini : 4
Is Johny alive : true
Caliing join()
Gini : 5
Johny : 1
Gini  terminated
Johny : 2
Johny : 3
Johny : 4
Johny : 5
Johny  terminated
Main Terminated
```

Beoz yield() was not there forcefull switching doesn't take place here, but natural switching can still take place

Note:

1. To invoke yield() and sleep() we do not require "**currentThread().**" This is because they are static methods and can be invoked without an instance(object).
2. Switching may take place even if yield() is removed. In that case, it will be natural switching.

d) setPriority(): public final void setPriority(int level)

In java, each thread is assigned a priority which affects the order in which it is scheduled for running. The threads of equal priority are given same treatment by java scheduler and therefore they share the processor at regular intervals.

The setPriority() method of java allows us to set priority for each thread between 1 and 10 with 1 being the minimum and 10 being the maximum priority. The Thread class also defines three constants which can be used along with this method. They are MIN_PRIORITY(1), NORM_PRIORITY(5) and MAX_PRIORITY(10).

By default all threads have NORM_PRIORITY of 5. By giving priorities to thread we can make sure a thread gets the attention it deserves. For example, if a thread is waiting for an input from the user then it should be given top priority.

When multiple threads are ready for execution java chooses that thread which has the highest priority and executes it. For a thread of lower priority to gain control one of the following things should happen:

a. It (Higher priority thread) dies naturally at the end of run() method.
b. It is made to sleep using sleep() method
c. It is told to wait() using wait method.

Also keep in mind that ***join() supersedes setPriority().***

→ Default priority of Thread = 5

Example: Usage of setPriority()

```
class MyRunnable implements Runnable
{
public void run()
{
        String  s = Thread.currentThread().getName();

        for(int i=1;i<=5;i++)
                System.out.println(s + " : " + i);

        System.out.println(s + "  terminated");
}
}
class ThreadPriority1
{
public static void main(String args[])
{
MyRunnable  r = new MyRunnable();

Thread t1 = new Thread(r , "Gini");
Thread t2 = new Thread(r , "Johny");

t1.setPriority(Thread.MIN_PRIORITY); // Same as 1
t2.setPriority(Thread.MAX_PRIORITY); // Same as (10)

t1.start();    t2.start();

try
{
  t1.join();     t2.join();
}
catch(InterruptedException e)
{  System.out.println(e);}
System.out.println("Main Terminated");
}}
```

Ideal Output: In single core CPU:

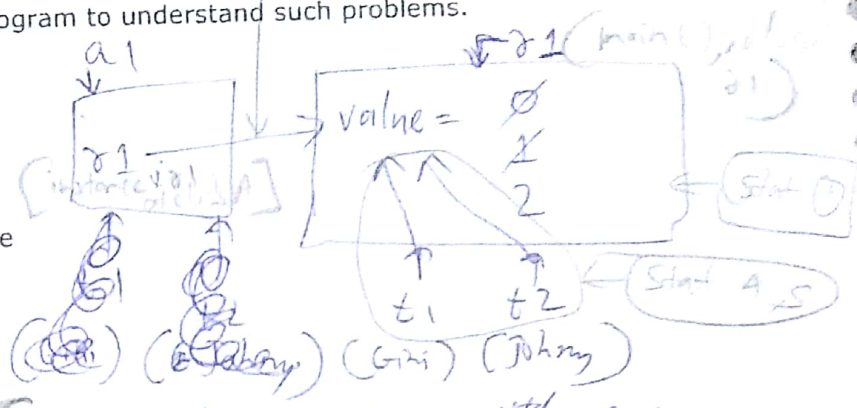
If there are more than 1 core CPU, then each core works on diff threads, so we will see jumbled O/P. So concept of setPriority() and join() will be seen only in single core CPU

```
Johny : 1
Johny : 2
Johny : 3
Johny : 4
Johny : 5
Johny  terminated
Gini : 1
Gini : 2
Gini : 3
Gini : 4
Gini : 5
Gini  terminated
Main Terminated
```

Note: Because join() supersedes setPriority(), thats why "Main Terminated" is displayed last although main's priority(5) is higher than t1(1).

Synchronization

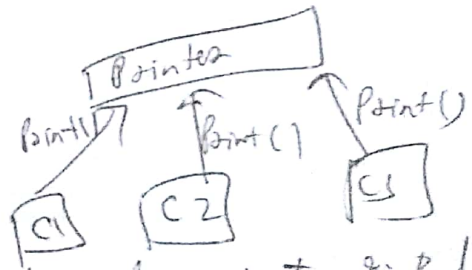
1. Sometimes it may be required that multiple threads need to access a single resource(for example, a single variable).
2. In that case, some care need to be taken to make sure that they don't access the same resource at the same time. Otherwise, we get unexpected output.
3. Consider the following program to understand such problems.



t1 and t2 are racing with each other to access resource 'value'

```
class Resource
{
        public int value ;
}

class A implements Runnable
{
        Resource r1 ;

        A(Resource x)
        {
                r1 = x;
        }

        public void run()
        {
                String s = Thread.currentThread().getName();
                System.out.println(s + " increments value by 1 ");
                ++ r1.value;
                // try { Thread.sleep(1000); } catch(InterruptedException e) { }
                System.out.println("Now value = " + r1.value);
        }
}

class WithoutSyn
{
public static void main(String args[])
{
Resource r1 = new Resource();
r1.value = 0;

A a1 = new A(r1);

Thread t1 = new Thread(a1 , "Gini");
Thread t2 = new Thread(a1 , "Johny");

t1.start();
t2.start();
}
}
```

Some diag how programs will pg



→ real life eg of Synchronisat^n :-
If 3 computers are not Synch, then the ~~output~~ printed papers can be mixed up.

Sandeep J. Gupta (9821882868)

4. The following are some of the outputs which you may get:

```
Gini increments value by 1
Johny increments value by 1
Now value = 1
Now value = 2
```

or

```
Johny increments value by 1
Gini increments value by 1
Now value = 1
Now value = 2
```

or

```
Gini increments value by 1
Johny increments value by 1
Now value = 2
Now value = 2
```

Note :-
@ This O/P will come if we write sleep() in between if there is switching after the statement "++gi.value"

The next program gives us the ideal output.

5. The above outputs demonstrate a problem which is known as a "race condition" where multiple threads can access the same resource (typically an object's instance variables) and can produce corrupted data. Here one thread is in the middle of a multi-step operation and another thread changes some value or condition on which the first thread was depending.
6. To avoid such "race condition" problems we must make sure that the threads access the single resource in some ***serialized manner***, that is, the functioning of these threads should be ***synchronized***. Given below is an example:

```
class Resource
{
      public int  value ;
}

class A implements Runnable
{
      Resource  r1 ;

      A(Resource x)
      {
            r1 = x;
      }

      public synchronized void run()
      {
            String  s = Thread.currentThread().getName();
            System.out.println(s + " increments value by 1 ");
            ++ r1.value;
            // try { Thread.sleep(1000); } catch(InterruptedException  e) { }
            System.out.println("Now value = " + r1.value);
      }
}
class UsingSyn
{
public static void main(String args[])
{
Resource  r1 = new Resource();
r1.value = 0;

A  a1 = new A(r1);

Thread  t1 = new Thread(a1 , "Gini");
Thread  t2 = new Thread(a1 , "Johny");

t1.start();
t2.start();
}
}
```

now there will be no switching in method run()

Output:

```
Gini increments value by 1
Now value = 1
Johny increments value by 1
Now value = 2
```

Note: Even if sleep() is present, we will get the same output.

7. Alternative to above program:

```
class Resource
{
        public int  value ;
}

class A implements Runnable
{
        Resource  r1 ;

        A(Resource x)
        {
                r1 = x;
        }

        public void run()
        {
                synchronized(this)
                {
                String  s = Thread.currentThread().getName();
                System.out.println(s + " increments value by 1 ");
                ++ r1.value;
                System.out.println("Now value = " + r1.value);
                }
        }
}
class ObjectSyn1
{
public static void main(String args[])
{
Resource  r1 = new Resource();
r1.value = 0;

A  a1 = new A(r1);

Thread  t1 = new Thread(a1 , "Gini");
Thread  t2 = new Thread(a1 , "Johny");

t1.start();
t2.start();
}
}
```

Output:

```
Gini increments value by 1
Now value = 1
Johny increments value by 1
Now value = 2
```

Note:

In this program we are using the concept of synchronized block. The working of this program and earlier program will be exactly same. Keep in mind that 'this' keyword stands for the invoking object which in this case is first t1 and then t2.

8. One more alternative

```
class Resource
{       public int  value ;   }

class A implements Runnable
{
        Resource  r1 ;

        A(Resource x)
        {               r1 = x;         }

        public void run()
        {
                String  s = Thread.currentThread().getName();
                System.out.println(s + " increments value by 1 ");

                synchronized(this)
                {
                ++ r1.value;
                System.out.println("Now value = " + r1.value);
                }
        }
}
class ObjectSyn2
{
public static void main(String args[])
{
Resource  r1 = new Resource();
r1.value = 0;

A  a1 = new A(r1);

Thread  t1 = new Thread(a1 , "Gini");
Thread  t2 = new Thread(a1 , "Johny");

t1.start();             t2.start();
}
}
```

This technique is used where we don't want switching between some statements but we want switching between some statements inside a function

We may now get the Output:

```
Gini increments value by 1
Johny increments value by 1
Now value = 1
Now value = 2
```

These 2 statements may occur in any order

But in the above program we will never get the output:

```
Gini increments value by 1
Johny increments value by 1
Now value = 2
Now value = 2
```

even if sleep() is present.

Thread Locks

1. Every object in Java has a built-in lock that only comes into play when the object has synchronized method code.
2. When we enter a synchronized non-static method, we automatically acquire the lock associated with the current instance of the class whose code we're executing (the this instance).
3. Acquiring a lock for an object is also known as getting the lock or locking the object or locking on the object or synchronizing on the object. We may also use the term monitor to refer to the object whose lock we're acquiring.
4. Since there is only one lock per object, if one thread has picked up the lock, no other thread can pick up the lock until the first thread releases the lock. This means no other thread can enter the synchronized code (which means it can't enter any synchronized method of that object) until the lock has been released.
5. Typically, releasing a lock means the thread holding the lock (in other words, the thread currently in the synchronized method) exits the synchronized method. At that point, the lock is free until some other thread enters a synchronized method on that object.

Example:

```
class Resource
{       public int  value ;  }

class A implements Runnable
{
        Resource  r1 ;

        A(Resource x)
        {               r1 = x;       }

        public void run()
        {
                synchronized(r1)
                {
                String  s = Thread.currentThread().getName();
                System.out.println(s + " increments value by 1 ");
                ++ r1.value;
                System.out.println("Now value = " + r1.value);
                }
        }
}
class ObjectLock
{
public static void main(String args[])
{
Resource  r1 = new Resource();          r1.value = 0;

A  a1 = new A(r1);

Thread  t1 = new Thread(a1 , "Gini");
Thread  t2 = new Thread(a1 , "Johny");

t1.start();             t2.start();
}}
```

Note :- When we write t1.start(), eventually method run() executes, Now because of code "synchronized(r1)", Thread t1 acquires a lock on that object which is pointed by r1. This means that now no other thread can work on that object till t1 releases the lock on that object (i.e when the syn block is completed)

(Difference on previous page, we wrote this here)

but when a syn block of one thread is completed, then the other Thread can acquire lock on same previous resource

t1 has acquired lock on r1

Output:
Gini increments value by 1
Now value = 1
Johny increments value by 1
Now value = 2

Note: In this program we may say that threads t1 and t2 acquire a lock on object r1 one after another.

Rules about locking and synchronization:

1. Only methods (or blocks) can be synchronized, not variables or classes.
2. Each object has just one lock.
3. Not all methods in a class need to be synchronized. A class can have both synchronized and non-synchronized methods.
4. If two threads are about to execute a synchronized method in a class and both threads are using the same instance of the class to invoke the method, only one thread at a time will be able to execute the method. The other thread will need to wait until the first one finishes its method call. In other words, once a thread acquires the lock on an object, no other thread can enter any of the synchronized methods in that class (for the object).
5. If a class has both synchronized and non-synchronized methods, multiple threads can still access the class's non-synchronized methods.
6. If a thread goes to sleep, it still holds any locks it has - it doesn't release them.
7. A thread can acquire more than one lock. For example, a thread can enter a synchronized method, thus acquiring a lock, and then immediately invoke a synchronized method on a different object, thus acquiring that lock as well.
8. You can synchronize a block of code rather than a method.

Thread-safe Classes

1. When a class has been carefully synchronized to protect its data, we say the class is "thread-safe".
2. Many classes in the Java APIs already use synchronization internally in order to make the class "thread-safe".
3. For example, StringBuffer and StringBuilder are nearly identical classes, except that all the methods in stringBuffer are synchronized, while those in StringBuilder are not.
4. Generally, this makes StringBuffer safe to use in a multithreaded environment, while StringBuilder is not.
5. In return, StringBuilder is a little bit faster than StringBuffer.

(Bcoz, thread safe classes run slower)

Deadlock

Deadlock occurs when two threads are blocked, with each waiting for the other's lock. Neither can run until the other gives up its lock, so they'll sit there forever.

Example 1: Demonstrates deadlock

We wanted to show deadlock, so we wanted a lock on r1 by t1 which is achieved only in Synchronized Blocks so we wrote syn(r1), syn(r2)

```
class Resource
{
      public int  value ;
}

class A implements Runnable
{
      Resource  r1 , r2 ;

      A(Resource x , Resource y)
      {
            r1 = x;
            r2 = y;
      }

      public void run()
      {
            synchronized(r1)
            {
                  System.out.println("A locked on r1 ");
                  System.out.println(r1.value);

                  // try { Thread.sleep(1000); } catch(InterruptedException  e) { }
                  // sometimes required

                  System.out.println("A waiting to lock on r2 ");

                  synchronized(r2)
                  {
                        System.out.println("A locked on r2 ");
                        System.out.println(r2.value);
                  }
            }
      }
}
```

Expected O/P:-
A locked on r1
5
A waiting to lock on r2
A locked on r2
6
B locked on r2
6
B waiting to lock on r1
B locked on r1
5

Actual O/P
B locked on r2
A locked on r1
6
5
B waiting to lock on r1
A waiting to lock on r2
[And then the program hangs due to deadlock]

```
class B implements Runnable
{
        Resource  r1 , r2 ;

        B(Resource x , Resource y)
        {
                r1 = x;
                r2 = y;
        }

        public void run()
        {
                synchronized(r2)
                {
                        System.out.println("B locked on r2 ");
                        System.out.println(r2.value);

                        // try { Thread.sleep(1000); } catch(InterruptedException e) {}
                        // sometimes required

                        System.out.println("B waiting to lock on r1 ");

                        synchronized(r1)
                        {
                                System.out.println("B locked on r1 ");
                                System.out.println(r1.value);
                        }
                }
        }
}


class Deadlock1
{
public static void main(String args[])
{
Resource  r1 = new Resource();
Resource  r2 = new Resource();

r1.value = 5;
r2.value = 6;

A  a1 = new A(r1,r2);
B  b1 = new B(r1,r2);

Thread  t1 = new Thread(a1);
Thread  t2 = new Thread(b1);

t1.start();
t2.start();
}
}
```

Note:
To avoid deadlock in the above example, the ORDER of acquiring object locks should be SAME. Refer next program.

Scanned by CamScanner

Example 2: Avoiding Deadlock

Even if sleep() is uncommented, then too locks are not released

```
class Resource
{       public int  value ;  }

class A implements Runnable
{
        Resource  r1 , r2 ;

        A(Resource x , Resource y)
        {               r1 = x;                 r2 = y;         }

        public void run()
        {
            synchronized(r1)
            {
                System.out.println("A locked on r1 ");
                System.out.println(r1.value);
                // try { Thread.sleep(1000); } catch(InterruptedException  e) { }
                System.out.println("A waiting to lock on r2 ");

                synchronized(r2)
                {
                    System.out.println("A locked on r2 ");
                    System.out.println(r2.value);
                }
            }
        }
}

class B implements Runnable
{
        Resource  r1 , r2 ;

        B(Resource x , Resource y)
        {               r1 = x;                 r2 = y;         }

        public void run()
        {
            synchronized(r1)
            {
                System.out.println("B locked on r1 ");
                System.out.println(r1.value);
                // try { Thread.sleep(1000); } catch(InterruptedException  e) { }
                System.out.println("B waiting to lock on r2");

                synchronized(r2)
                {
                    System.out.println("B locked on r2");
                    System.out.println(r1.value);
                }
            }
        }
}
```

Even if sleep is uncommented, lock will not release

Consider first two :- T_1 has $o1$, then switching, T_2 wants $o1$, but cannot have it, also T_2 cannot have $o2$, as Syn($o2$) block is inside block Syn($o1$). Here Deadlock is not possible

```
class Deadlock2
{
public static void main(String args[])
{
Resource  r1 = new Resource();
Resource  r2 = new Resource();

r1.value = 5;
r2.value = 6;

A  a1 = new A(r1,r2);
B  b1 = new B(r1,r2);

Thread  t1 = new Thread(a1);
Thread  t2 = new Thread(b1);

t1.start();
t2.start();
}
}
```

Output:-
A locked on r1
5
A waiting to lock on r2
A locked on r2
6
B locked on r2 & r1
5
B waiting to lock on r2
B locked on r2
6

Thread Interaction

1. The **Object** class has three methods – wait(), notify() and notifyAll() – using which threads can communicate with each other.

   - **wait()** tells the calling thread to give up the monitor(object lock) and go to sleep until some other thread enters the same monitor and calls **notify( )**.
   - **notify()** wakes up the first thread that called **wait()** on the same object.
   - **notifyAll()** wakes up all the threads that called **wait()** on the same object. The highest priority thread will run first. notifyAll() must be used when several threads are waiting on one object and we want to be sure that all the waiting threads get notified. If multiple threads are waiting on the same object and if we use notify() instead of notifyAll() then **ONLY ONE** thread will get notified and others will keep waiting forever.

2. These three methods ***must*** be called from within a ***synchronized context*** only.
3. A thread can't invoke a wait or notify method on an object unless it owns that object's lock.
4. When the wait method is invoked on an object, the thread executing that code gives up its lock on the object immediately.
5. However, when notify is called, that doesn't mean that the thread gives up its lock at that moment. If the thread is still completing synchronized code, the lock is not released until the thread moves out of synchronized code. So just because notify() is called, this doesn't mean that the lock becomes available at that moment.

V. Imp

[notify() should be written after wait()]

Example 1:

```
import java.util.*;

class Wait1 implements Runnable
{
      Calculator c;

      public Wait1(Calculator  calc)
      {
            c = calc ;
      }

      public void run()
      {
            synchronized(c)
            {
                  Scanner sc = new Scanner(System.in);
                  System.out.print("Enter n: ");
                  c.n = sc.nextInt();
                  System.out.println("Waiting for result");

                try
                  {
                     //Thread.sleep(1000);
                     c.wait();
                  }
                  catch(InterruptedException e)
                  {  System.out.println(e);    }

                  System.out.println("Sum is " + c.sum);
            }
      }

      public static void main(String args[])
      {
            Calculator cal = new Calculator();
            Wait1  w = new Wait1(cal);

            Thread t1 = new Thread(cal);
            Thread t2 = new Thread(w);

            t2.start();
            t1.start();
            // Note we started t2 first
      }
}
```

Step 1 → t2 acquired lock on 'c'

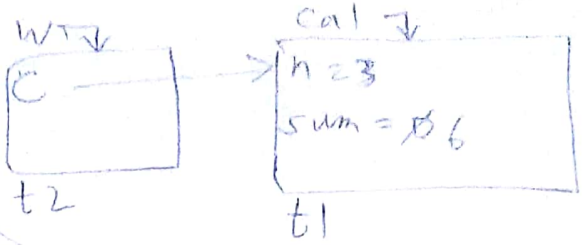
Should be same

because first wait() should execute then notify().

Note - Let's say, there are 2 more threads, t3 and t4 and we have the statement c.wait() within threads t2, t3, t4. This means that the 3 threads are waiting on the same object 'c'. In this case, if we write notify() within thread t1, then the Thread Scheduler will resume execution of only 1 of the 3 threads. The remaining 2 threads keep waiting forever. [So in this situation we should write notifyAll()]

→ If notify() would have been invoked by some other object (not c), then thread t2 would have kept waiting forever.


```
class Calculator implements Runnable
{
    int n, sum=0;
    public void run()
    {
        synchronized(this)
        {
            for(int i=1;i<=n;i++)
                sum = sum + i;
            //try { Thread.sleep(1000); } catch(InterruptedException e) { }
            notify();
            System.out.println("Let me complete synchronized code !!");
        }
    }
}
```



Note :- this means invoking obj which is t2. and t1 is associated with cal

Output:

```
Enter n: 4
Waiting for result
Let me complete synchronized code !!
Sum is 10
```

What is the importance of third line of output ?

It proves that the thread will not release the block as soon as it sees notify(). Lock will be released only on completion of synchronized block.

What happens if we interchange order of start() ?

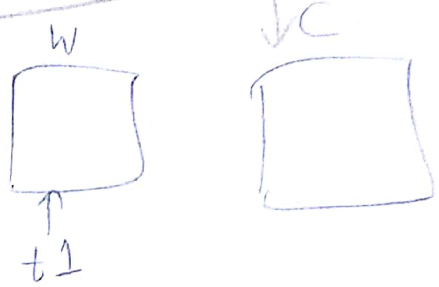
In this case, the run() of class Calculator will execute first as a result of which notify() may execute before wait(). Hence the program hangs. Bcoz thread t2 will wait infinitely as for the lock to be released.

What happens if we put sleep() ?

No change in O/P.

Example 2: IllegalMonitorStateException

```
class WaitException1 implements Runnable
{
        public void run()
        {
                synchronized(this)
                {
                    try
                        {
                            WaitException1  c = new WaitException1();
                            c.wait();
                        }
                        catch(InterruptedException e)
                        { }
                }
        }

        public static void main(String args[])
        {
                WaitException1  w = new WaitException1();
                Thread t1 = new Thread(w);
                t1.start();
        }
}
```



On running this program we get:
Exception in thread "Thread-0" java.lang.IllegalMonitorStateException

Why exception is thrown?
Because wait() is invoked on object c but c does not own the lock.
The lock is owned by "this"

Example 3 : using wait and synchronized within static context

```
class WaitException2 implements Runnable
{
        public static void main(String args[])
        {
                synchronized(this)
                {
                        try
                        {
                                wait();
                        }
                        catch(InterruptedException e)
                        { }
                }
        }
}
```

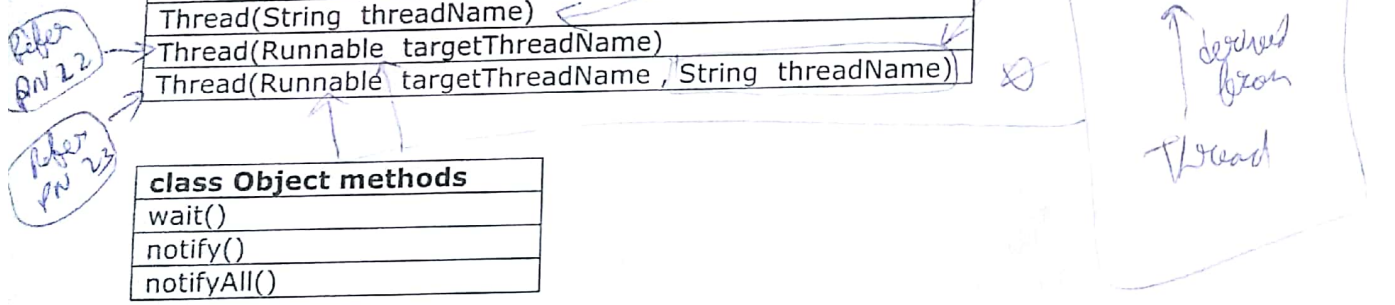
Two Errors:

1. non-static variable this cannot be referenced from a static context : synchronized(this).
2. non-static method wait() cannot be referenced from a static context : wait();
   This means that wait() must be invoked using an object.

Summary of important Methods of this Chapter

class Thread Methods	
start()	Starts a thread by calling run()
run()	Entry point for thread
getName()	Returns name of thread
setName()	Sets name of thread
getId ()	Returns a positive, unique, long number, and that number will be that thread's ID number for the thread's entire life.
setPriority()	Sets the thread priority
getPriority()	Returns the thread priority
yield()	Used to relinquish (hand over) control to another thread
isAlive()	Returns true if the calling thread is alive else false
sleep()	Causes thread to sleep for specified no. of milliseconds.
join()	Sometimes it is necessary for one thread to wait until another thread dies. This is done using the join method.

class Thread constructors
Thread()
Thread(String threadName)
Thread(Runnable targetThreadName)
Thread(Runnable targetThreadName , String threadName)



class Object methods
wait()
notify()
notifyAll()

javac and java command

1. The **javac** command is used to invoke Java's compiler.
2. The syntax of javac command is: **javac [options] [source files]**
3. Both the [options] and the [source files] are optional parts of the command, and both allow multiple entries.
4. For the exam, you'll need to understand the –classpath (-cp) and -d options.
5. The invocation 'javac' without any 'options' or 'source files' is valid. It doesn't compile any files, but prints a summary of valid options as shown below:

```
Usage: javac <options> <source files>
where possible options include:
  -g                         Generate all debugging info
  -g:none                    Generate no debugging info
  -g:{lines,vars,source}     Generate only some debugging info
  -nowarn                    Generate no warnings
  -verbose                   Output messages about what the compiler is doing
  -deprecation               Output source locations where deprecated APIs are
used
  -classpath <path>          Specify where to find user class files and
annotation processors
  -cp <path>                 Specify where to find user class files and annotation
processors
  -sourcepath <path>         Specify where to find input source files
  ...                        ...                        ...                        ...
```

6. The **java** command is used to invoke Java Virtual Machine.
7. The syntax of java command is: **java [options] class [args]**
8. Both the [options] and the [args] are optional parts of the command, and both allow multiple entries.
9. For the exam, you'll need to understand the –classpath (-cp) and -D options.

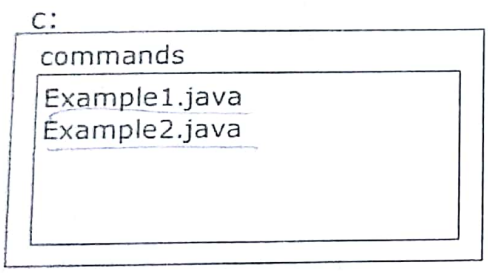
Steps for execute pgm :-
1) cd ...
2) set path ---/bin
3) javac xyz.java
4) java xyz

Throughout the following topics we will use these two example programs:

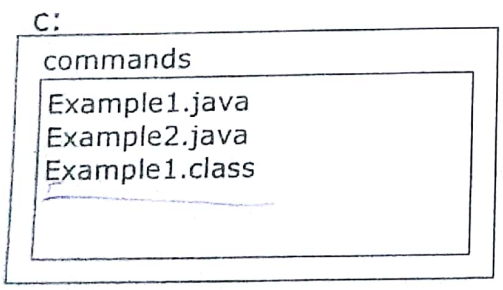
```
1)
class Example1
{
public static void main(String[ ] args)
{
  System.out.println("Example1") ;
}
}

2)
class Example2
{
public static void main(String[ ] args)
{
  System.out.println("Example2") ;
}
}
```

A) Compiling and Executing a Single Program

Lets say that the current folder is: C:\commands
and
We have the following



Compile as: **c:\commands>** javac Example1.java

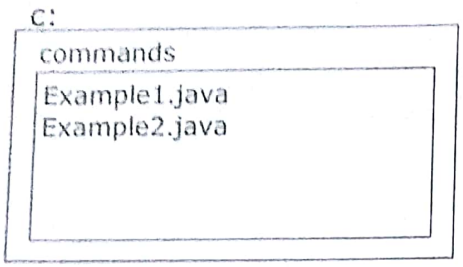


Execute as: **c:\commands>**java Example1

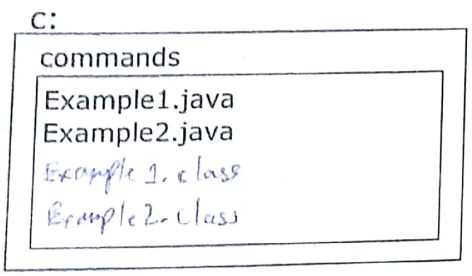
Output: Example1

B) Compiling and Executing multiple programs

Lets say that the current folder is: C:\commands
and
We have the following



Compile as: **c:\commands>**javac Example1.java Example2.java



Execute as: **c:\commands>**java Example1 Example2

Output: Example1

Execute as: **c:\commands>**java Example2 Example1

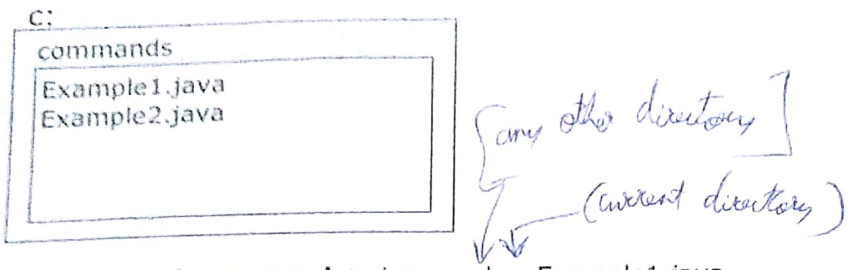
Output: Example2

C) –d option while compiling & -cp while running

1. d stands for destination.
2. By default, the compiler puts a .class file in the same directory as the .java source file.
3. This is fine for very small projects, but once you're working on a project of big size, you'll want to keep your .java files separated from your .class files. This helps with version control, testing, deployment etc.
4. The -d option lets you tell the compiler in which directory to put the .class file(s).

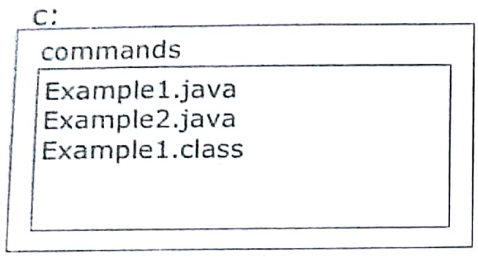
i) Compiling and Executing in the same directory

Lets say that the current folder is: C:\commands
and
We have the following



[any other directory]
(current directory)

Compile as: **c:\commands>** javac -d . Example1.java
(There should be at least one space before and after dot)

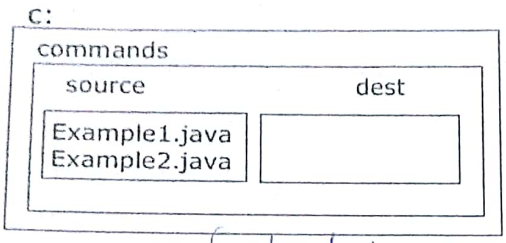


Execute as: **c:\commands>**java Example1

Output: Example1

ii) Compiling and Executing from a different directory

Lets say that the current folder is: **c:\commands**
and
We have the following



Compile as:

c:\commands>javac -d dest source\Example1.java

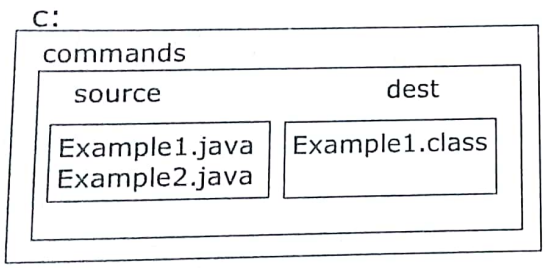
OR

c:\commands>javac -d c:\commands\dest c:\commands\source\Example1.java

OR

c:\commands>javac -d c:/commands/dest c:/commands/source/Example1.java

- Notice the difference in slash.



- If we remove the underlined part then Example1.class will be created in 'source' directory.

Execute as:

c:\commands>java Example1
Error: Could not find or load main class Example1

c:\commands>java -cp c:\commands\dest Example1

OR

c:\commands>java -classpath c:\commands\dest Example1

Output: Example1

Wrong Command: **c:\commands>**java -cp c:\commands\dest\Example1

Possible Mistakes:

1) c:\commands>javac **-D** c:/commands/dest c:/commands/source/Example1.java
javac: invalid flag: -D
'd' cannot be given in uppercase

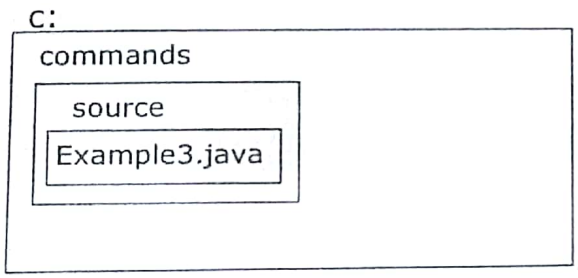
2) c:\commands>javac -d Example1.java
javac: directory not found: Example1.java
You must mention a path along with –d option

(. is also valid after -d as . is for the path of current directory)

iii) Package

Example 1:

Lets say that the current folder is: **c:\commands**
and
We have the following

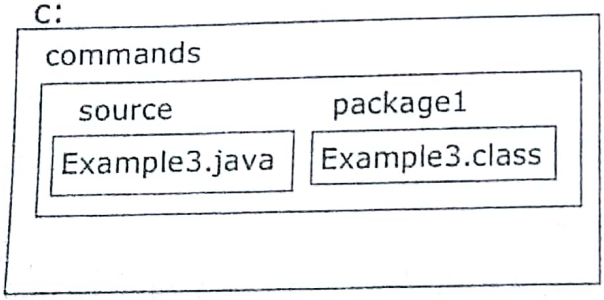



```
package package1;
class  Example3
{
public  static  void  main(String[ ]  args)
{
  System.out.println("Example3")
}
}
```

[In Javac, this is Source]

{This is destination}

Compile as: **c:\commands>**javac -d **.** source\Example3.java

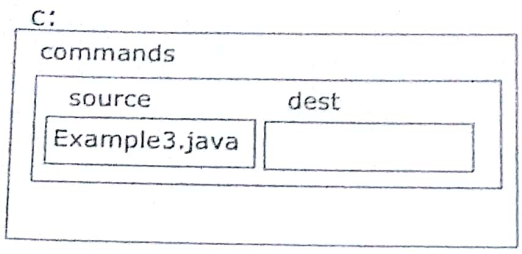


Execute as: **c:\commands>**java -cp c:\commands package1.Example3

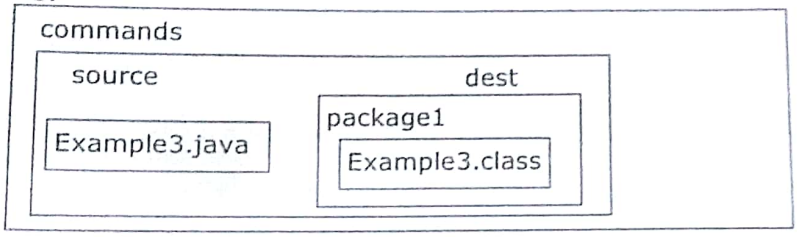
Output: Example3

Example 2:

Lets say that the current folder is: **c:\commands**
and
We have the following



Compile as:
c:\commands>javac -d c:\commands\dest c:\commands\source\Example3.java



Execute as: **c:\commands>**java -cp c:\commands\dest package1.Example3

Output: Example3

Example 3:

Lets say that the current folder is: **c:\myproject\source**
and
We have the following java file

```
package com.wickedlysmart;
class  MyClass
{
public  static  void  main(String[ ]  args)
{
  System.out.println("MyClass") ;
}
}
```

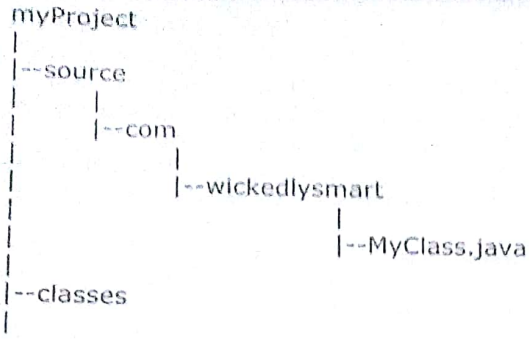
and the following directory structure

```
myproject
|
|--source
|       |
|       |--com
|               |
|               |--wickedlysmart
|                               |
|                               |--MyClass.java
|
|--classes
|       |
|       |--com
|               |
|               |--wickedlysmart
|                               |
|                               |-- (MyClass.class should come here)
```

Compile as:
c:\myproject\source>javac -d ..\classes com\wickedlysmart\MyClass.java

The above command could be read: "To set the destination directory, cd back to the myProject directory then cd into the classes directory, which will be your destination. Then compile the file named MyClass.java. Finally, put the resulting MyClass.class file into the directory structure that matches its package, in this case, classes/com/wickedlysmart." Because MyClass.java is in a package, the compiler knew to put the resulting .class file into the classes/com/wickedlysmart directory.

Somewhat amazingly, the javac command can sometimes help you out by building directories it needs! Suppose we have the following directory structure:



And the following command (the same as last time):
c:\myproject\source>javac -d ..\classes com\wickedlysmart\MyClass.java

In this case, the compiler will build two directories called com and com/wickedlysmart in order to put the resulting MyClass.class file into the correct package directory (com/wickedlysmart/) which it builds within the existing .../classes directory.

The last thing about -d that you'll need to know for the exam is that if the destination directory you specify doesn't exist, you'll get a compiler error.

If, in the previous example, the classes directory did NOT exist, the compiler would say something like:
java:5: error while writing MyClass: classes/MyClass.class (No such file or directory)

Execute as:

c:\myproject\source>java -cp ..\classes com.wickedlysmart.MyClass

OR

c:\myproject\source>java -cp c:\myproject\classes com.wickedlysmart.MyClass

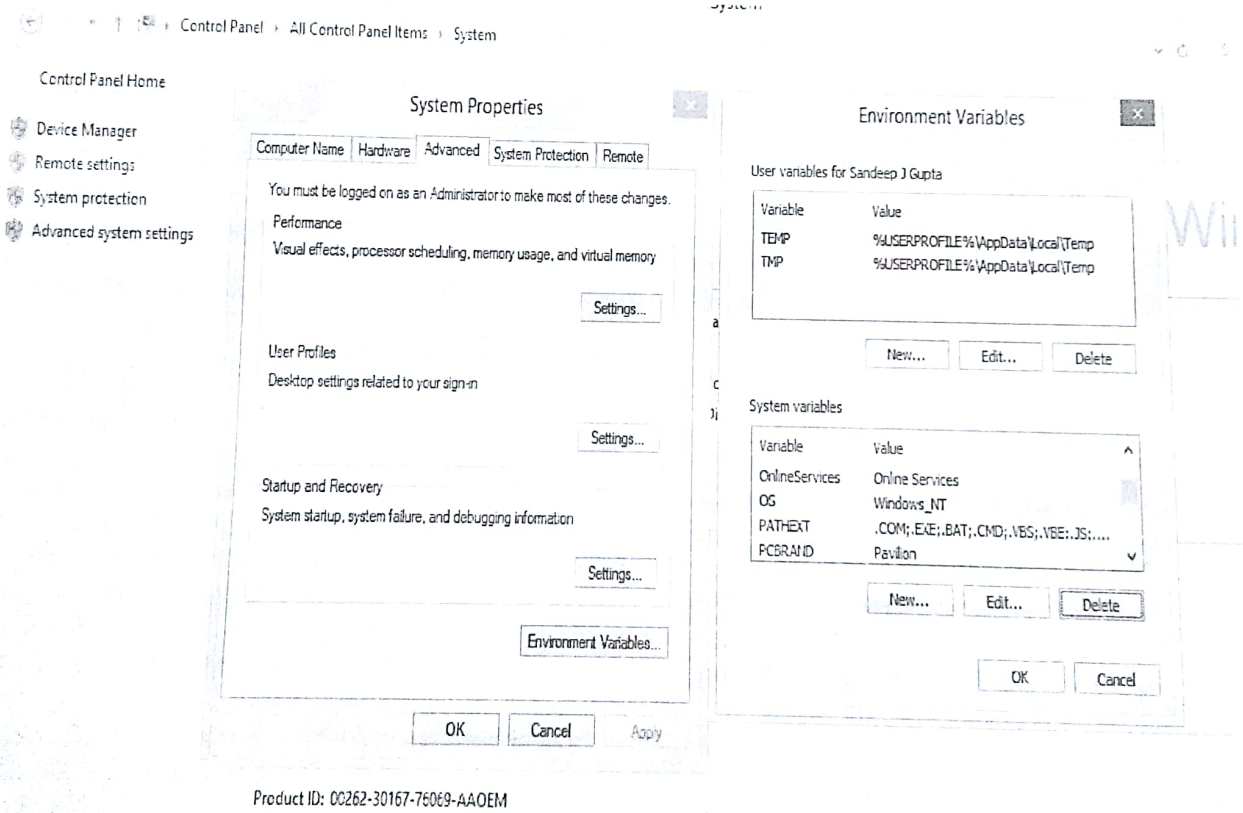
Output:
MyClass

Classpaths

- When we use the java and javac commands, we want these commands to search for classes that will be necessary to complete the required operation.
- Both java and javac use the same basic search algorithm. They both have the same list of places (directories) which they search for classes. They both search through this list of directories in the same order.
- As soon as they find the class they're looking for, they stop searching for that class. In case their search list contains two or more files with the same name, the first file found will be used to complete the required operation.
- The first place they look is in the directories that contain the classes that come standard with J2SE.
- The second place they look is in the directories defined by classpaths. Classpaths should be thought of as "class search paths." They are lists of directories in which classes might be found.
- There are two places where classpaths can be declared – Environment Variable and Command Line.

A) Environment Variable

- A classpath can be declared as an operating system environment variable.
- The classpath declared here is used, by default, whenever java or javac is invoked.



B) Command Line:

- A classpath can be declared as a command-line option for either java or javac.
- *Classpaths declared as command-line options override the classpath declared as an environment variable, but they persist only for the length of the program invocation.*

Declaring and Using Classpaths at Command Line:

1. A classpath consists of a variable number of directory locations, separated by delimiters. For Unix-based operating systems, forward slashes are used to construct directory locations, and the separator is the colon (:). For example,
   -classpath /com/foo/acct:/com/foo
   specifies two directories in which classes can be found - /com/foo/acct & /com/ foo. It's important to remember that when you specify a subdirectory, you're NOT specifying the directories above it. For instance, in the preceding example the directory /com will NOT be searched.

2. ***Most of the path-related questions on the exam will use Unix conventions. If you are a Windows user, your directories will be declared using backslashes (\) and the separator character you use will be a semicolon (;).***

3. A very common situation occurs in which java or javac complains that it can't find a class file, and yet you can see that the file is in the ***current directory***! When searching for class files, the java and javac commands don't search the current directory by default. You must tell them to search there. The way to tell java or javac to search in the current directory is to add a ***dot (.)*** to the classpath.

4. For example, -classpath /com/foo/acct:/com/foo:**.**
   This classpath is identical to the previous one EXCEPT that the dot (.) at the end of the declaration instructs java or javac to *also* search for class files in the current directory.

5. It's also important to remember that classpaths are ***searched from left to right.*** Therefore in a situation where classes with same names are located in different directories different results will occur. Hence,

   -classpath /com:/foo:. is not the same as **-classpath .:/foo:/com**

6. Finally, also remember that the java command allows you to abbreviate -classpath as -cp.

Example:

Lets say that the current directory is: c:\commands
and we have the following directory structure:

```
commands
|
|--Demo.class
|
|--classpath
|       |
|       |--Demo.class
```

Class Demo in directory **commands** is as shown:

```
class  Demo
{
public  static  void  main(String[]  args)
{
  System.out.println("Within directory - commands") ;
}
}
```

Class Demo in directory **commands\classpath** is as shown:

```
class  Demo
{
public  static  void  main(String[]  args)
{
  System.out.println("Within directory - classpath") ;
}
}
```

Also there is one more class Demo in directory **c:\other** which is as shown:

```
class  Demo
{
public  static  void  main(String[]  args)
{
  System.out.println("Within directory - other") ;
}
}
```

Command Given at Command Line	Output
C:\commands>java Demo	Within directory - commands
C:\commands>java -cp . Demo	Within directory - commands
C:\commands>java -cp classpath Demo	Within directory - classpath
C:\commands>java -cp classpath\ Demo	Within directory - classpath
C:\commands>java -cp c:\other Demo	Within directory - other
C:\commands>java -cp c:\other;classpath;. Demo	Within directory - other
C:\commands>java -cp .;c:\other;classpath Demo	Within directory - commands
C:\commands>java -cp classpath;.;c:\other Demo	Within directory - classpath

Using System Properties

1. Java has a class called java.util.Properties that can be used to access a system's persistent information such as the current version of the operating system, the Java version, OS name etc.
2. In addition to providing such default information, you can also add and retrieve your own properties.
3. The class Properties has two important methods – getProperty() and setProperty().
4. The getProperty() method is used to retrieve a single property. It can be invoked with a single argument (a String that represents the name (or key)), or it can be invoked with two arguments (a String that represents the name (or key) and a default String value to be used as the property if the property does not already exist). In both cases, getProperty() returns the property as a String.
5. The setProperty() method is used for setting system property. System properties can also be set at command line using –D option.

Example of getProperty()

```
import java.util.*;

class  GetProperty
{
public  static  void  main(String[ ]  args)
{
 Properties p = System.getProperties();
 String s;

 s = p.getProperty("java.version");
 System.out.println(s) ;

 s = p.getProperty("java.vendor");
 System.out.println(s) ;

 s = p.getProperty("java.home");
 System.out.println(s) ;

 s = p.getProperty("user.name");
 System.out.println(s) ;

 s = p.getProperty("os.name");
 System.out.println(s) ;

 s = p.getProperty("os.version");
 System.out.println(s) ;

 s = p.getProperty("file.separator");
 System.out.println(s) ;

 s = p.getProperty("path.separator");
 System.out.println(s) ;
}
}
```

Example1 of setProperty() : updating already existing system property

```
import java.util.*;

class  SetProperty1
{
public  static  void  main(String[ ]  args)
{
 Properties p = System.getProperties();
 String s;

 System.out.println(p.getProperty("user.name")) ;

 p.setProperty("user.name","RameshBabu");
 System.out.println(p.getProperty("user.name")) ;

 System.out.println(p.getProperty("java.vendor")) ;
}
}
```

Output, when command is: java SetProperty1
Sandeep J Gupta
RameshBabu
Oracle Corporation

Output, when command is: java -Djava.vendor="Study Circle" SetProperty1
Sandeep J Gupta
RameshBabu
Study Circle

Note:
Double quotes compulsory when there are spaces. We could have put java.vendor also in double quotes but its not required, since there are no spaces.

Example2 of setProperty() : adding a new property

```
import java.util.*;
class  SetProperty2
{
public  static  void  main(String[ ]  args)
{

 Properties p = System.getProperties();

 p.setProperty("myProp", "myValue");
 p.list(System.out);
 System.out.println("******************") ;
 System.out.println(p.getProperty("cmdProp")) ;
}
}
```

Output, when command is: java SetProperty2

```
...     ...     ...
os.name=Mac OS X
myProp=myValue
...     ...     ...
******************
null
```

Output, when command is: java -DcmdProp=cmdValue SetProperty2

```
...     ...     ...
os.name=Mac OS X
myProp=myValue
...     ...     ...
******************
cmdValue
```

Note:

1. See the difference in the last line of output.

2. setProperty() can be used to add and retrieve your own properties.

3. The ... in the output represent lots of other name=value pairs. (The *name* and *value* are sometimes called the *key* and the *property*.) Two name=value properties were added to the system's properties: myProp=myValue was added via the setProperty method, and cmdProp=cmdVal was added via the -D option at the command line. When using the -D option, if your value contains white space the entire value should be placed in quotes.

4. Just in case you missed it, when you use -D, the name=value pair must follow *immediately*, no spaces allowed.

Encapsulation, Coupling & Cohesion

The following are some of the goals of Object Oriented (OO) programming
* Ease of creation
* Ease of maintenance
* Ease of enhancement
* Reusability and
* Reliability.

Tight encapsulation, Loose coupling and High cohesion help to achieve the above goals of OO design.

A) Encapsulation

Encapsulation refers to combining of data(variables or fields) and methods (that manipulate this data) in a single unit called class.

A class could be tightly encapsulated or loosely (weakly) encapsulated.

In Tight encapsulation, no fields of an object can be modified or accessed directly. You can only access the fields through a method call.

Hence in a tightly encapsulated class all the instance variables (fields) are private and the class contains 'getter' (accessor) and 'setter' (mutator) methods.

The most important benefit of Tight Encapsulation is that you can monitor and validate all changes done to a field.

Example of Loose Encapsulation

```
class Employee
{
  public String name;
  public double salary;
        ...     ...
        ...     ...
}

Within User of class Employee:

Employee e = new Employee();
e.salary = 28000;
```

Note:
In the "Tight Encapsulation" example, the employee's salary is modified after some validation. This is not possible in case of Loose Encapsulation.

Example of Tight Encapsulation

```
class Employee
{
  private String name;
  private double salary;

  double getSalary() {return salary;}

  public void setSalary(double  newSalary)
  {
    if(newSalary<0)
     System.out.println("Can't be negative");
    else if(newSalary<salary)
      System.out.println("New Salary lesser
                          than existing salary");
    else
      salary = newSalary;
  }
        ...     ...
        ...     ...
}

Within User of class Employee:
Employee e = new Employee();
e.setSalary(28000);
```

B) Coupling

1. Coupling (or dependency) is defined as the degree of interdependence between two or more modules (classes, functions etc).
2. Two modules are considered independent if one can function completely without the presence of other. However, all the modules in a system cannot be independent of each other, as they must interact with each other so that together they produce the desired output.
3. For example, in a program to implement a queue we have a function remove() which removes an element from the queue. remove() makes use of another function empty() which checks whether the queue is empty or not. It is now obvious that function remove() is coupled with function empty().
4. In OOP, it is desirable that modules should be ***loosely coupled***. Loose coupling gives following benefits:
   a. A module can be completely understood without knowing other modules.
   b. Making changes is easier since changes in one module do not affect others.

C) Cohesion

1. Cohesion refers to how closely related the specific tasks of a module (class, function etc) are.
2. Ideally a module should be ***highly cohesive***. High cohesion occurs when a module performs a set of closely related tasks.
3. In other words, it is not desirable that the components in a module perform tasks which are quite unrelated with each other. That would be low cohesion.
4. For example, any sensible programmer would create two different classes Stack and Queue to implement data structures stack and queue respectively. Here, a bad programming practice would be to create a single class, say *DataStructure*, which performs all the operations related to data structures stack and queue. We can now say that this class is not at all cohesive.
5. Classes that implement high cohesion are more reusable & easier to test and debug.

Example of Low Cohesion

```
class StudentResult
{
  private  String name;
  private  int  roll , no_of_subjects , total;
  private  int  marks[];
  private  double  percentage;

  private int  bookId;
  private int  no_of_days;
  private double  libraryFine;

  public void readMarks()
  { ... }

  public void calculatePercentage()
  {...}

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

Note:
The class StudentResult exhibits low cohesion since it contains members which perform two different tasks. One set of members deal with calculation of total marks and another set of members deal with computation of library fine.

Ideally there should be a separate class to deal with library related tasks.