



Yadab Raj Ojha

Sr. Java Developer / Lecture

Email: yadabrajojha@gmail.com

Blog: <http://yro-tech.blogspot.com/>

Java Tutorial: <https://github.com/yrojha4ever/JavaStud>

LinkedIn: <https://www.linkedin.com/in/yrojha>

Twitter: <https://twitter.com/yrojha4ever>

Part 4

Multithreading:

- Multithreading in Java
- Thread and process
- Thread Life Cycle
- Joining Thread, Thread Priority
- Thread Pool, Thread Group
- Shutdown Hook, Garbage Collection
- Synchronization, Runtime class

File IO

- File Management
- File Streams
- Data Streams
- Stream Tokenization
- Random Access Files
- Short IO Projects
- File Class
- Scanner Class

Serialization

Deserialization

Multithreading in Java

Multithreading in java is a process of executing multiple threads simultaneously.

Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation etc.

Process vs. threads

A *process* runs independently and isolated of other processes. It cannot directly access shared data in other processes. The resources of the process, e.g. memory and CPU time, are allocated to it via the operating system.

A *thread* is a so called lightweight process. It has its own call stack, but can access shared data of other threads in the same process. Every thread has its own memory cache. If a thread reads shared data it stores this data in its own memory cache. A thread can re-read the shared data.

A Java application runs by default in one process. Within a Java application you work with several threads to achieve parallel processing or asynchronous behavior.

Advantage of Java Multithreading

- 1) It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.
- 2) You **can perform many operations together so it saves time**.
- 3) Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:

- Process-based Multitasking(Multiprocessing)
- Thread-based Multitasking(Multithreading)

1) Process-based Multitasking (Multiprocessing)

- Each process have its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

2) Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- Thread is lightweight.
- Cost of communication between the thread is low.

What is Thread in java

A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.

Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.

According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated.

How to create thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread. JVM calls the run() method on the thread.
3. **public void sleep(long milliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long milliseconds):** waits for a thread to die for the specified milliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(deprecated).
16. **public void resume():** is used to resume the suspended thread(deprecated).
17. **public void stop():** is used to stop the thread(deprecated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

Runnable interface:

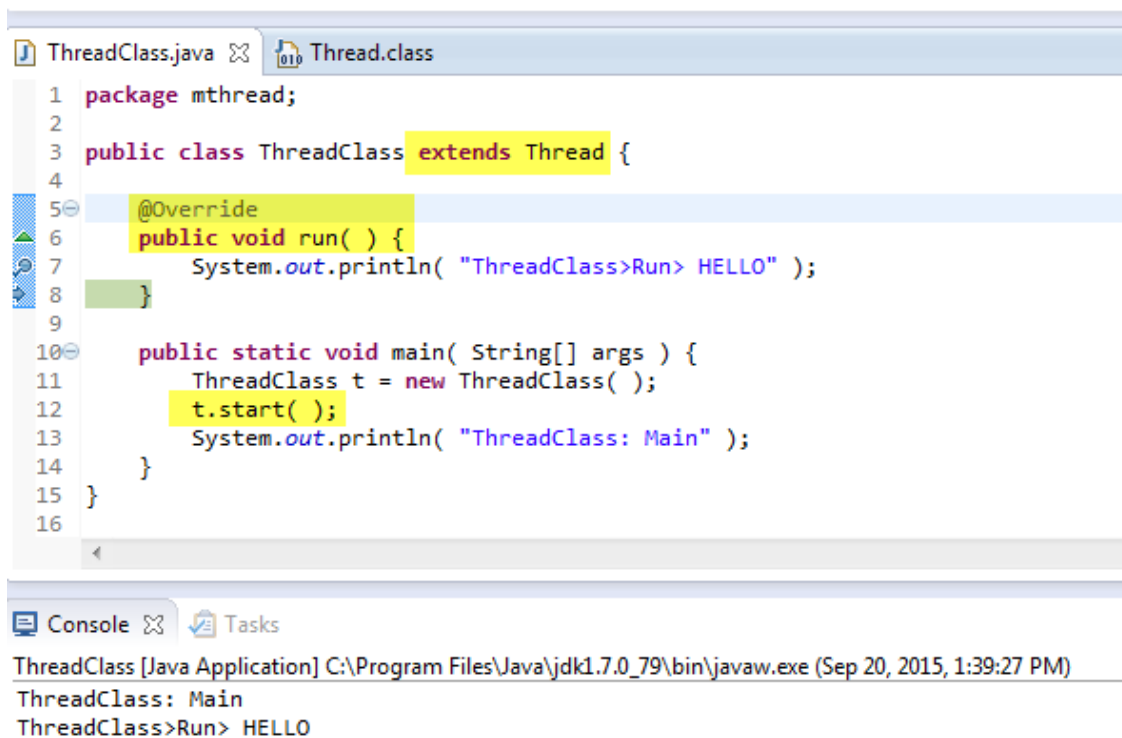
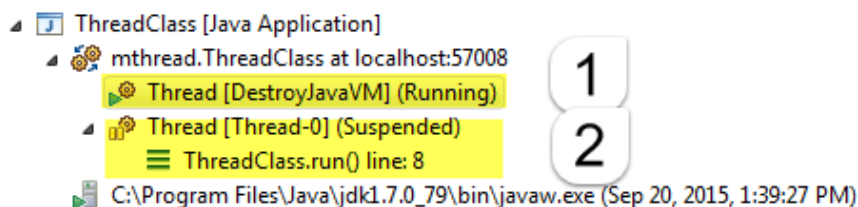
The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

1. **public void run():** is used to perform action for a thread.

Starting a thread:

start() method of Thread class is used to start a newly created thread. It performs following tasks:

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.



RunnableInterface [Java Application]

- hread.ThreadRunnableInterface at localhost:57052
 - Thread [DestroyJavaVM] (Running)
 - Thread [Thread-0] (Suspended)

C:\Program Files\Java\jdk1.7.0_79\bin\javaw.exe (Sep 20, 2015, 1:47:18 PM)

```

1 package mthread;
2
3 public class RunnableInterface implements Runnable {
4
5     @Override
6     public void run( ) {
7         System.out.println( "Thread: Runnable> run()" );
8     }
9
10    public static void main( String[] args ) {
11        RunnableInterface runnable = new RunnableInterface( );
12        Thread t = new Thread( runnable );
13        t.start( );
14        System.out.println( "Thread: RunnableInterface> Main()" );
15    }
16
17 }
18

```

Console Tasks

RunnableInterface [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\javaw.exe (Sep 20, 2015, 1:47:18 PM)

Thread: RunnableInterface> Main()

Thread: Runnable> run()

Thread Scheduler in Java

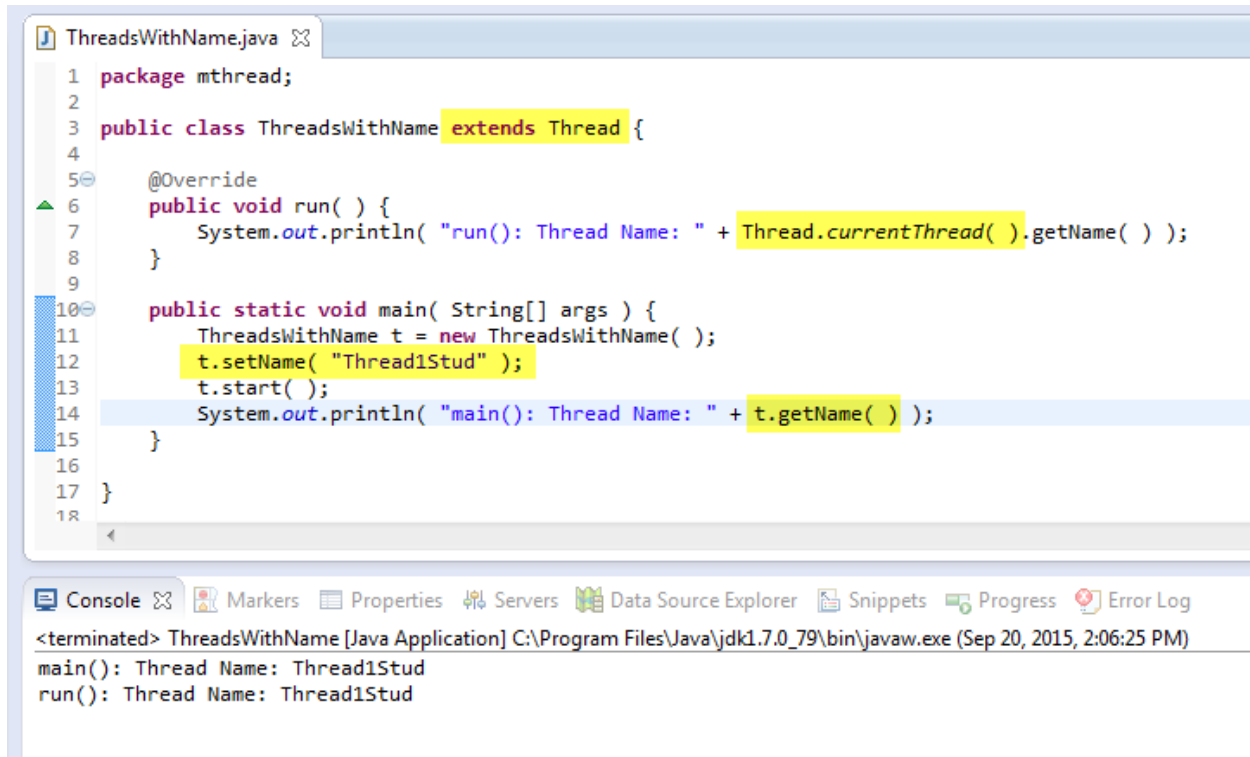
Thread scheduler in java is the part of the JVM that decides which thread should run.

There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.

Only one thread at a time can run in a single process.

The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

The currentThread() method & naming a Thread:



```
1 package mthread;
2
3 public class ThreadsWithName extends Thread {
4
5     @Override
6     public void run( ) {
7         System.out.println( "run(): Thread Name: " + Thread.currentThread( ).getName( ) );
8     }
9
10    public static void main( String[] args ) {
11        ThreadsWithName t = new ThreadsWithName( );
12        t.setName( "Thread1Stud" );
13        t.start( );
14        System.out.println( "main(): Thread Name: " + t.getName( ) );
15    }
16
17 }
```

Console Output:

```
<terminated> ThreadsWithName [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\javaw.exe (Sep 20, 2015, 2:06:25 PM)
main(): Thread Name: Thread1Stud
run(): Thread Name: Thread1Stud
```

Thread Scheduler in Java

Thread scheduler in java is the part of the JVM that decides which thread should run.

There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.

Only one thread at a time can run in a single process.

The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

Thread Priority

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

```
thread.setPriority( Thread.MAX_PRIORITY );
```


Thread.currentThread().getPriority()

Daemon Thread in Java

Daemon thread in java is a service provider thread that provides services to the user thread. Its life depends on the mercy of user threads i.e. when all the user threads die, JVM terminates this thread automatically.

- It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
- Its life depends on user threads.
- It is a low priority thread.

No.	Method	Description
1)	public void setDaemon(boolean status)	is used to mark the current thread as daemon thread or user thread.
2)	public boolean isDaemon()	is used to check that current is daemon.

Thread.Sleep() method in java:

- public static void sleep(long milliseconds) throws InterruptedException

The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

Thread.sleep(500); will halt thread for 500 milli second.

➤ Thread can not be started twice

java.lang.IllegalThreadStateException will be thrown.

➤ Do not call Run() method instead of Start() method.

Each thread starts in a separate call stack.

Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

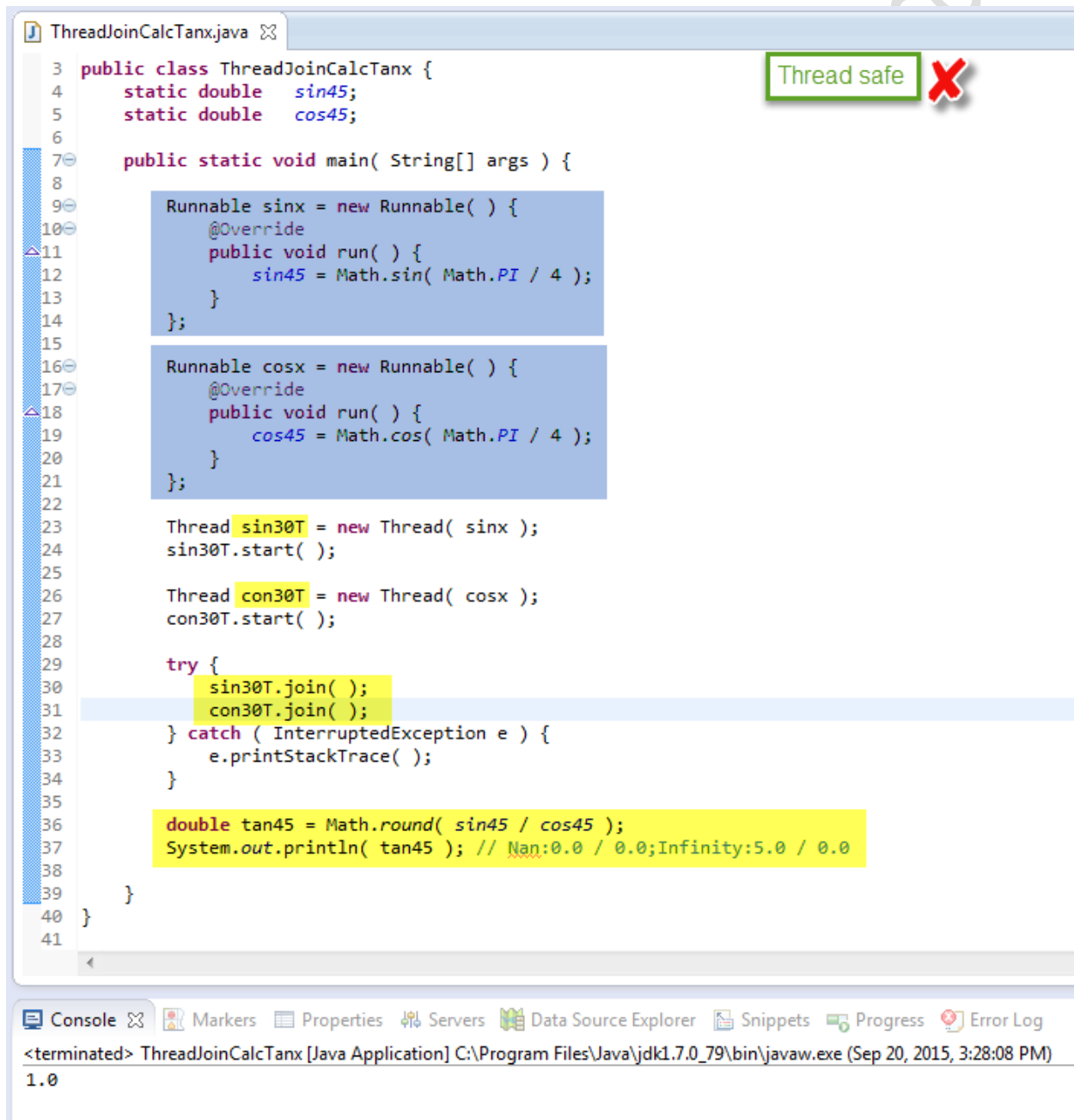
there is no context-switching because here thread1 and thread2 will be treated as normal object not thread object.

Thread.join() method:

The **join() method waits for a thread to die**. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

Example: Calculate $\tan(x) = \sin(x)/\cos(x)$

Where sin and cos value calculation should be completed before we calculate tanx



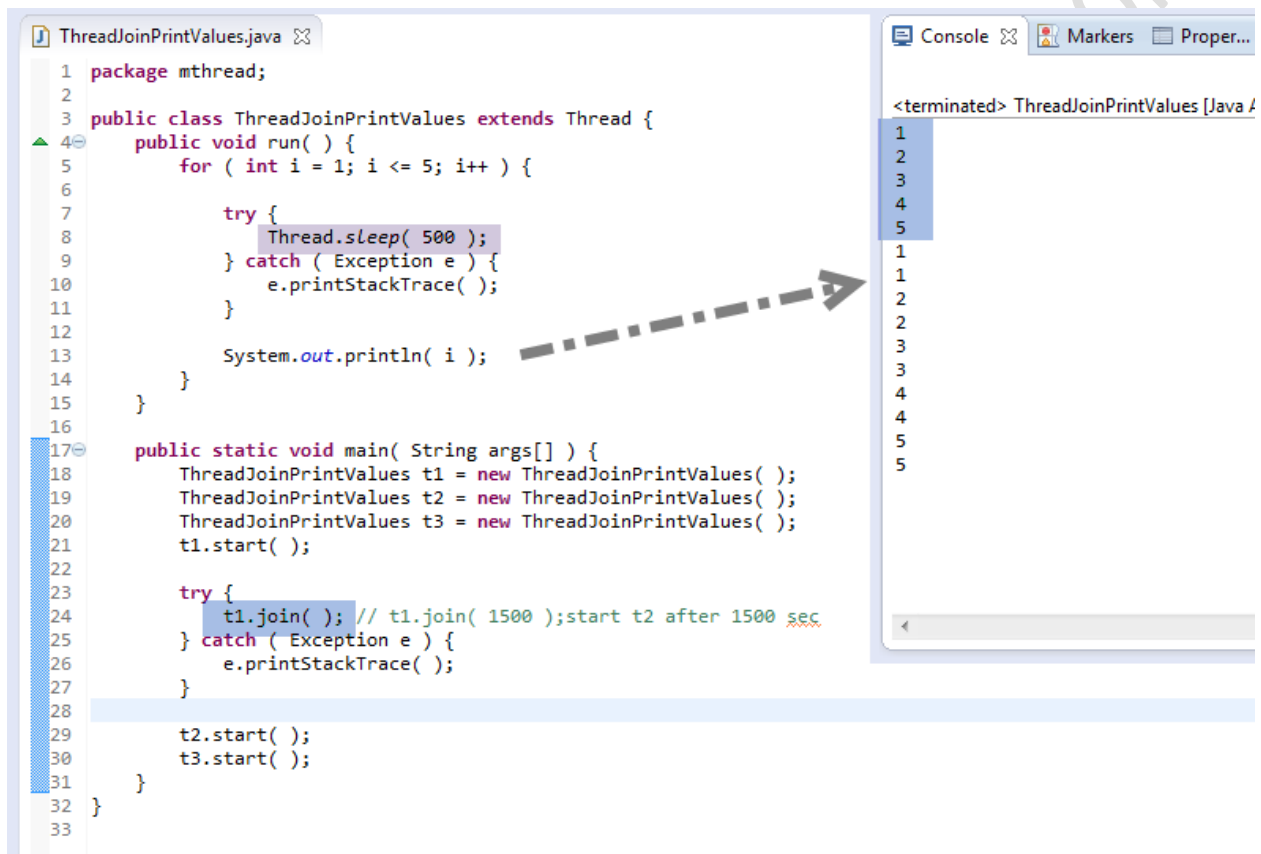
```
ThreadJoinCalcTanx.java
3 public class ThreadJoinCalcTanx {
4     static double sin45;
5     static double cos45;
6
7     public static void main( String[] args ) {
8
9         Runnable sinx = new Runnable( ) {
10             @Override
11             public void run( ) {
12                 sin45 = Math.sin( Math.PI / 4 );
13             }
14         };
15
16         Runnable cosx = new Runnable( ) {
17             @Override
18             public void run( ) {
19                 cos45 = Math.cos( Math.PI / 4 );
20             }
21         };
22
23         Thread sin30T = new Thread( sinx );
24         sin30T.start( );
25
26         Thread con30T = new Thread( cosx );
27         con30T.start( );
28
29         try {
30             sin30T.join( );
31             con30T.join( );
32         } catch ( InterruptedException e ) {
33             e.printStackTrace( );
34         }
35
36         double tan45 = Math.round( sin45 / cos45 );
37         System.out.println( tan45 ); // Nan:0.0 / 0.0;Infinity:5.0 / 0.0
38
39     }
40 }
41
```

Thread safe X

Console: <terminated> ThreadJoinCalcTanx [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\javaw.exe (Sep 20, 2015, 3:28:08 PM) 1.0

(example by anonymous class that implements Runnable interface. We can also make anonymous class that extend Thread class)

```
Thread t = new Thread( ) {  
    public void run( ) {  
        System.out.println( "Hello from thread." );  
    }  
};  
t.start( );
```



Shutdown Hook

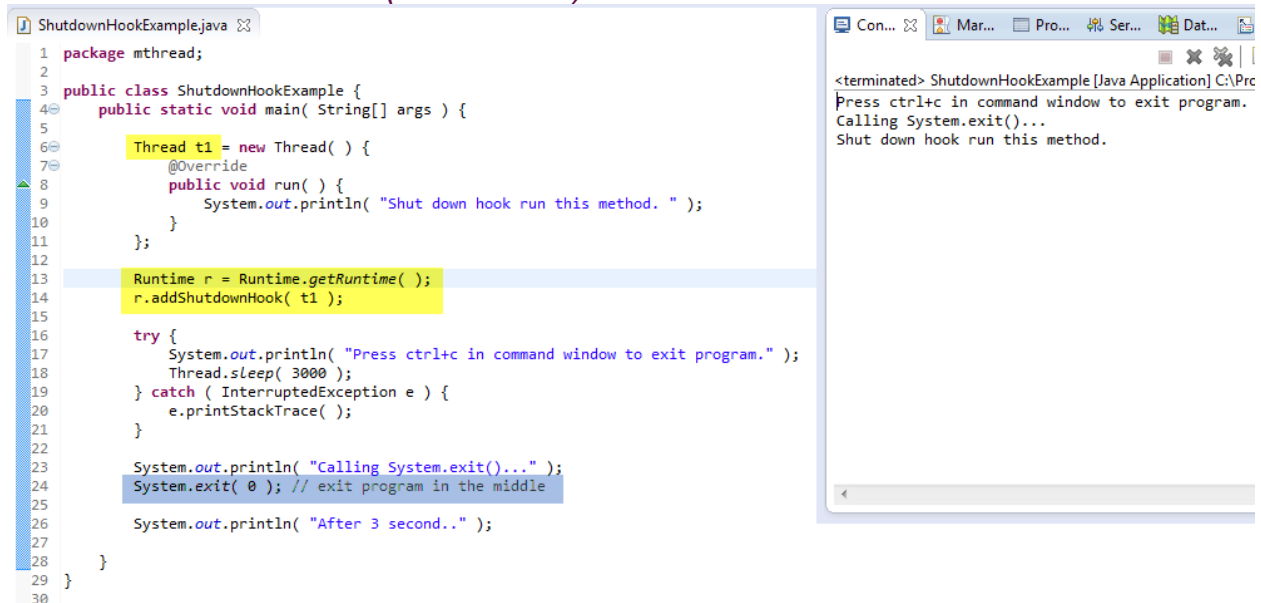
The shutdown hook can be used to perform cleanup resource or save the state when JVM shuts down normally or abruptly. Performing clean resource means closing log file, sending some alerts or something else. So if you want to execute some code before JVM shuts down, use shutdown hook.

When does the JVM shut down?

The JVM shuts down when:

- user presses ctrl+c on the command prompt
- System.exit(int) method is invoked
- user logoff
- user shutdown etc.

The addShutdownHook(Runnable r) method



Java Thread Pool

Java Thread pool represents a group of worker threads that are waiting for the job and reuse many times.

Better performance It saves time because there is no need to create new thread.

It is used in Servlet and JSP where container creates a thread pool to process the request.

```
ExecutorService executor = Executors.newFixedThreadPool(5);
```

```
executor.execute(r); //Runnable r
```

Thread Group in Java

Every Java thread is a member of a *thread group*. Thread groups provide a mechanism for collecting multiple threads into a single object and manipulating those threads all at once, rather than individually. For example, you can start or suspend all the threads within a group with a single method call. Java thread groups are implemented by the [ThreadGroup](#)^{api} class in the `java.lang` package.

```
ThreadGroup myThreadGroup = new ThreadGroup("My Group of Threads");
Thread myThread = new Thread(myThreadGroup, "a thread for my group");
```

Don't use `ThreadGroup` for new code. Use the `Executor` stuff in [java.util.concurrent](#) instead.

Synchronization in Java

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.

Example: When you have two threads that are reading and writing to the same 'resource', say a variable named `foo`, you need to ensure that these threads access the variable in an atomic way. Without the `synchronized` keyword, your thread 1 may not see the change thread 2 made to `foo`, or worse, it may only be half changed. This would not be what you logically expect.

```
package mthread;

public class SynchronizeFbLike {

    public static void main( String[] args ) {

        /* Facebook Page: Everest, Current Likes: 500 */
        final FacebookLike everestFbPagePiclike = new FacebookLike( 500 );

        Thread user1 = new Thread( ) {
            public void run( ) {
                everestFbPagePiclike.plusOne( );
            }
        };

        Thread user2 = new Thread( ) {
            public void run( ) {
                everestFbPagePiclike.plusOne( );
            }
        };

        Thread user3 = new Thread( ) {
            public void run( ) {
                everestFbPagePiclike.plusOne( );
            }
        };

        Thread user4 = new Thread( ) {
            public void run( ) {
                everestFbPagePiclike.plusOne( );
            }
        };

        /* User1,2,3,4 hit like in Everest Facebook Page */
        user1.start( );
        user2.start( );
        user3.start( );
        user4.start( );
    }
}
```

```
package mthread;

public class FacebookLike {

    public Integer likes = 0;

    /* set current Page likes */
    public FacebookLike( Integer likes ) {
        this.likes = likes;
    }

    /* Synchronized method call solve problem of Multi-thread problem */
    public synchronized void plusOne( ) {
        likes++;
        System.out.println( Thread.currentThread( ).getName( ) + " Likes: " + likes );
        try {
            Thread.sleep( 100 );
        } catch ( InterruptedException e ) {
            e.printStackTrace( );
        }
    }
}
```

Console Output:

Thread	Likes
Thread-0	501
Thread-2	502
Thread-3	503
Thread-1	504

Without synchronized keyword in above method

Java I/O

Java I/O (Input and Output) is used to process the input and produce the output based on the input.

Java uses the concept of stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

We can perform **file handling in java** by java IO API.

Stream

A stream is a sequence of data. In Java a stream is composed of bytes. It's called a stream because it's like a stream of water that continues to flow.

In java, 3 streams are created for us automatically. All these streams are attached with console.

1) System.out: standard output stream

2) System.in: standard input stream

3) System.err: standard error stream

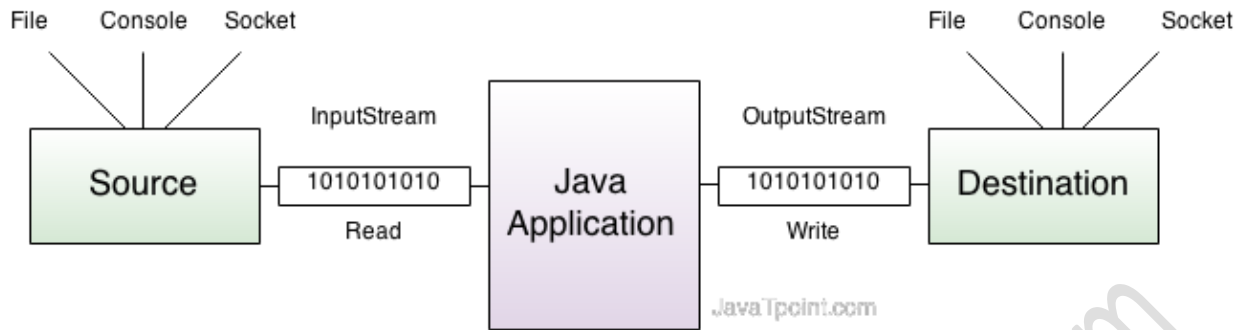
OutputStream

Java application uses an output stream to write data to a destination, it may be a file, an array, peripheral device or socket.

InputStream

Java application uses an input stream to read data from a source, it may be a file, an array, peripheral device or socket.

Let's understand working of Java OutputStream and InputStream by the figure given below.

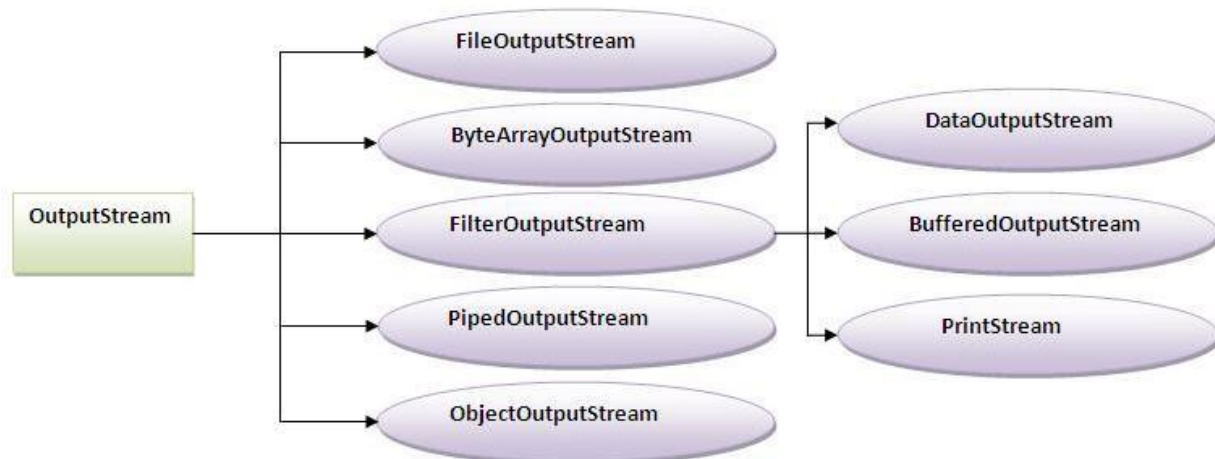


OutputStream class

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Commonly used methods of OutputStream class

Method	Description
1) public void write(int) throws IOException:	is used to write a byte to the current output stream.
2) public void write(byte[]) throws IOException:	is used to write an array of byte to the current output stream.
3) public void flush() throws IOException:	flushes the current output stream.
4) public void close() throws IOException:	is used to close the current output stream.

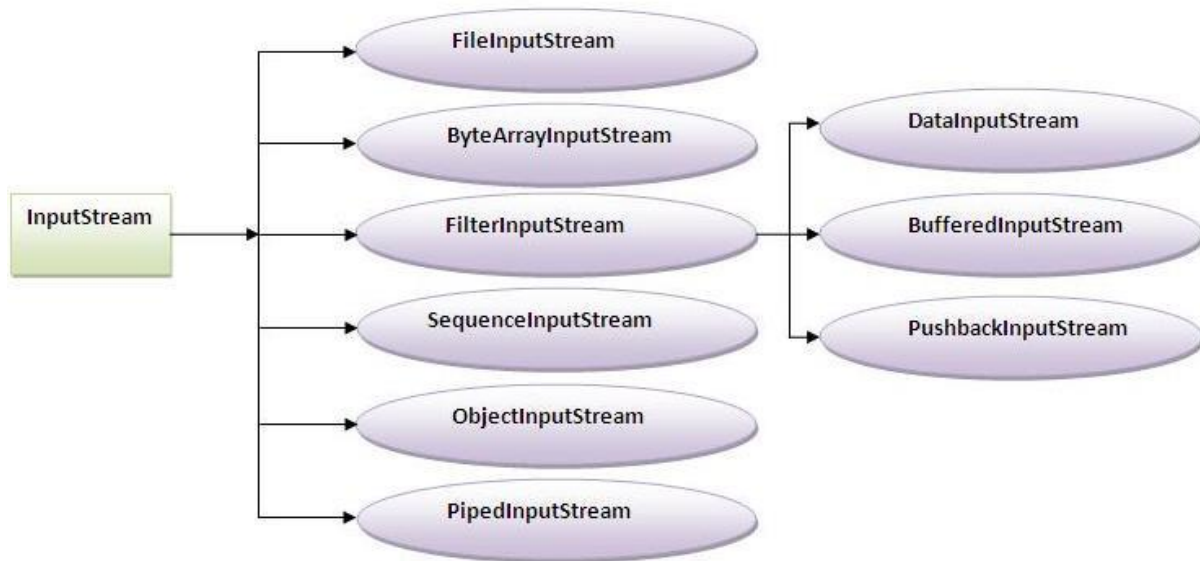


InputStream class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

Commonly used methods of InputStream class

Method	Description
1) public abstract int read()throws IOException:	reads the next byte of data from the input stream. It returns -1 at the end of file.
2) public int available()throws IOException:	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException:	is used to close the current input stream.



File Handling

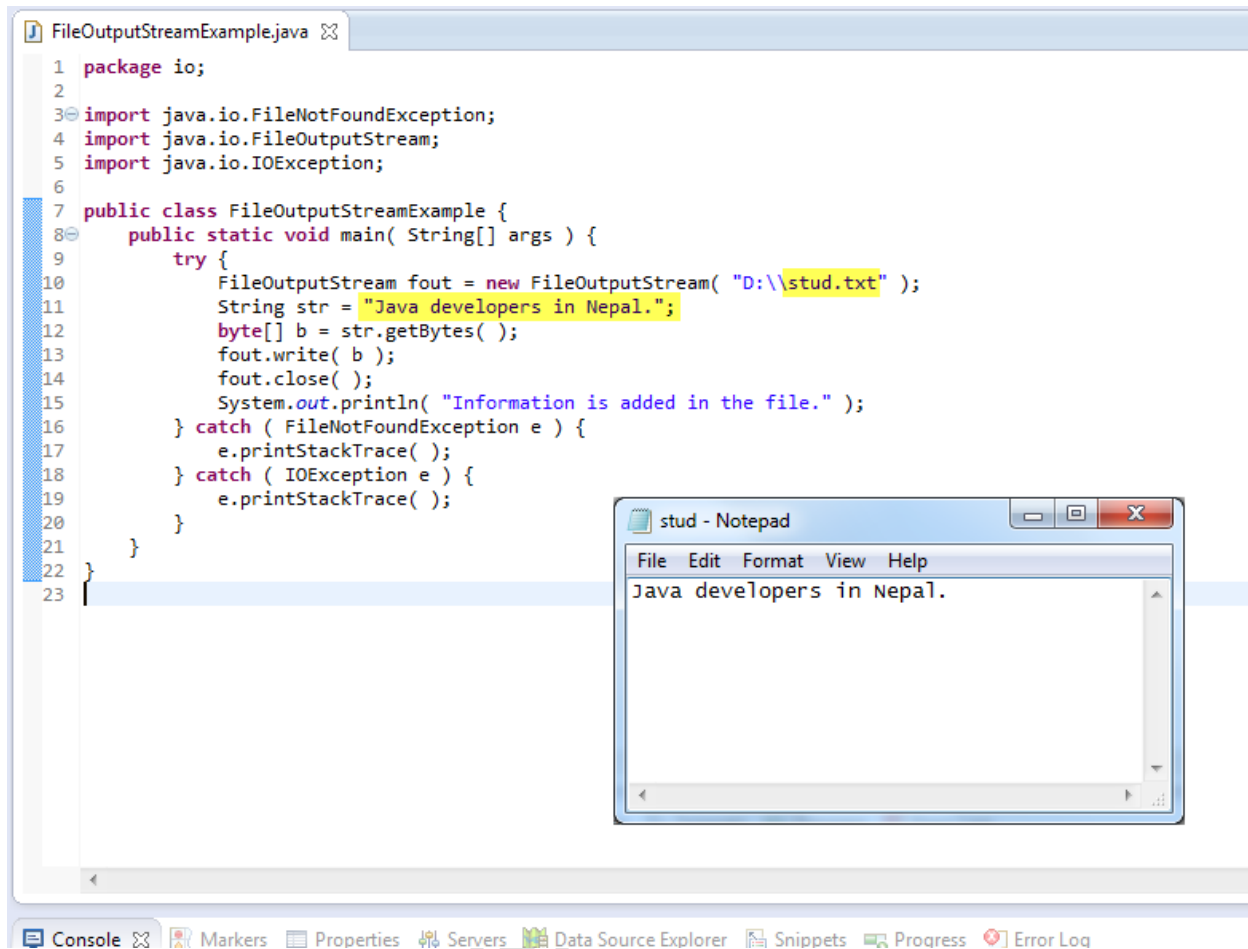
FileInputStream and FileOutputStream

In Java, `FileInputStream` and `FileOutputStream` classes are used to read and write data in file. In another words, they are used for file handling in java.

Java `FileOutputStream` class

Java `FileOutputStream` is an output stream for writing data to a file.

If you have to write primitive values then use `FileOutputStream`. Instead, for character-oriented data, prefer `FileWriter`. But you can write byte-oriented as well as character-oriented data.



Java FileInputStream class

Java FileInputStream class obtains input bytes from a file. It is used for reading streams of raw bytes such as image data. For reading streams of characters, consider using FileReader.

It should be used to read byte-oriented data for example to read image, audio, video etc.

```
FileStreamReadOneWriteToOther.java
1 package io;
2
3 import java.io.FileInputStream;
4 import java.io.FileOutputStream;
5
6 /**
7  * Copy content of one file to another.
8  * @author yojha
9  */
10 public class FileStreamReadOneWriteToOther {
11     public static void main( String[] args ) {
12
13         try {
14             FileInputStream fin = new FileInputStream( "D:\\stud.txt" );
15             FileOutputStream fout = new FileOutputStream( "D:\\stud2.txt" );
16             int i = 0;
17             while ( ( i = fin.read( ) ) != -1 ) {
18                 System.out.print( ( char ) i );
19                 fout.write( ( char ) i );
20             }
21             fin.close( );
22             fout.close( );
23
24         } catch ( Exception e ) {
25             e.printStackTrace( );
26         }
27     }
28 }
29
```

Java BufferedOutputStream and BufferedInputStream

Java BufferedOutputStream class

Java BufferedOutputStream class uses an internal buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

```
BufferedOutputStreamExample.java
1 package io;
2
3 import java.io.BufferedOutputStream;
4 import java.io.FileOutputStream;
5
6 public class BufferedOutputStreamExample {
7     public static void main( String[] args ) {
8         try {
9             FileOutputStream fout = new FileOutputStream( "D:\\bstud.txt" );
10            BufferedOutputStream bout = new BufferedOutputStream( fout );
11
12            String str = "Java developers of Nepal.\nRunning Chapter: File Handling, BufferedOutputStream";
13            byte[] b = str.getBytes( );
14
15            bout.write( b );
16            bout.flush( );
17
18            bout.close( );
19            fout.close( );
20
21            System.out.println( "Information is added in the file." );
22
23        } catch ( Exception e ) {
24            e.printStackTrace( );
25        }
26    }
27 }
28
```

Java BufferedInputStream class

Java BufferedInputStream class is used to read information from stream. It internally uses buffer mechanism to make the performance fast.

```
BufferedInputStreamExample.java
1 package io;
2
3 import java.io.BufferedInputStream;
4 import java.io.FileInputStream;
5
6 public class BufferedInputStreamExample {
7     public static void main( String args[] ) {
8         try {
9
10             FileInputStream fin = new FileInputStream( "D:\\bstud.txt" );
11             BufferedInputStream bin = new BufferedInputStream( fin );
12
13             int i;
14             while ( ( i = bin.read() ) != -1 ) {
15                 System.out.println( ( char ) i );
16             }
17
18             bin.close( );
19             fin.close( );
20
21         } catch ( Exception e ) {
22             e.printStackTrace( );
23         }
24     }
25 }
26
```

Java FileWriter and FileReader (File Handling in java)

Java FileWriter and FileReader classes are used to write and read data from text files. These are character-oriented classes, used for file handling in java.

Java has suggested not to use the FileInputStream and FileOutputStream classes if you have to read and write the textual information.

```
FileWriterExample.java
1 package io;
2
3 import java.io.FileWriter;
4
5 public class FileWriterExample {
6     public static void main( String args[] ) {
7         try {
8
9             FileWriter fw = new FileWriter( "D:\\fstud.txt" );
10
11             fw.write( "Java developers of नेपाल.\nRunning Chapter: File Handling, FileWriter." );
12
13             fw.close( );
14
15         } catch ( Exception e ) {
16             e.printStackTrace( );
17
18             System.out.println( "Information is added in the file." );
19         }
20     }
21 }
```

Make sure you have saved source file as UTF-8

```
FileReaderExample.java
1 package io;
2
3 import java.io.FileReader;
4
5 public class FileReaderExample {
6     public static void main( String args[] ) throws Exception {
7
8         FileReader fr = new FileReader( "D:\\fstud.txt" );
9         int i;
10
11         while ( ( i = fr.read( ) ) != -1 ) {
12
13             System.out.print( ( char ) i );
14         }
15
16         fr.close( );
17     }
18 }
19 }
```

Make sure you have saved source file as UTF-8

Console

<terminated> FileReaderExample [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\javaw.exe (Sep 20, 2016 10:10:10 AM)
Java developers of नेपाल.
Running Chapter: File Handling, FileWriter.

Reading data from keyboard

There are many ways to read data from the keyboard. For example:

- InputStreamReader
- Console
- Scanner
- DataInputStream etc.

InputStreamReader class

InputStreamReader class can be used to read data from keyboard. It performs two tasks:

- connects to input stream of keyboard
- converts the byte-oriented stream into character-oriented stream

BufferedReader class

BufferedReader class can be used to read data line by line by readLine() method.

reading data from keyboard by InputStreamReader and
BufferedReader class

```
BufferedReaderInputStreamReader.java
1 package io;
2
3 import java.io.BufferedReader;
4 import java.io.IOException;
5 import java.io.InputStreamReader;
6
7 public class BufferedReaderInputStreamReader {
8     public static void main( String[] args ) {
9
10         /**
11          * InputStreamReader in = new InputStreamReader( System.in );
12          * BufferedReader br = new BufferedReader(in);
13          */
14         BufferedReader br = new BufferedReader( new InputStreamReader( System.in ) );
15
16         try {
17             System.out.println( "Write anything." );
18
19             String name = br.readLine();
20
21             System.out.println( "You Wrote: " + name );
22
23         } catch ( IOException e ) {
24             e.printStackTrace();
25         }
26     }
27 }
28
29
```

Console

<terminated> BufferedReaderInputStreamReader [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\javaw.exe (Sep 20, 2015, 10:4

Write anything.

NEPAL

You Wrote: NEPAL

Java Scanner class

There are various ways to read input from the keyboard, the `java.util.Scanner` class is one of them.

The **Java Scanner** class breaks the input into tokens using a delimiter that is whitespace by default. It provides many methods to read and parse various primitive values.

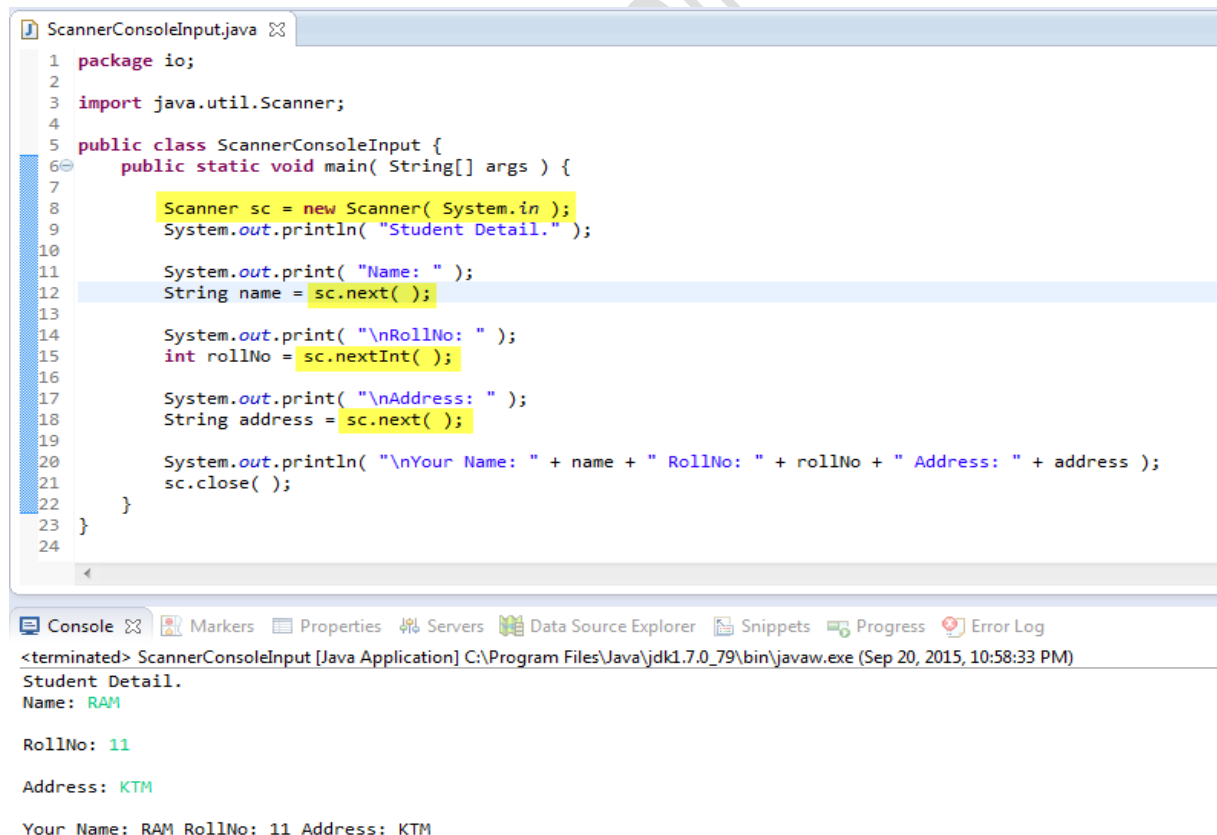
Java Scanner class is widely used to parse text for string and primitive types using regular expression.

Java Scanner class extends `Object` class and implements `Iterator` and `Closeable` interfaces.

Commonly used methods of Scanner class

There is a list of commonly used Scanner class methods:

Method	Description
public String next()	it returns the next token from the scanner.
public String nextLine()	it moves the scanner position to the next line and returns the value as a string.
public byte nextByte()	it scans the next token as a byte.
public short nextShort()	it scans the next token as a short value.
public int nextInt()	it scans the next token as an int value.
public long nextLong()	it scans the next token as a long value.
public float nextFloat()	it scans the next token as a float value.
public double nextDouble()	it scans the next token as a double value.



```
1 package io;
2
3 import java.util.Scanner;
4
5 public class ScannerConsoleInput {
6     public static void main( String[] args ) {
7
8         Scanner sc = new Scanner( System.in );
9         System.out.println( "Student Detail." );
10
11         System.out.print( "Name: " );
12         String name = sc.next( );
13
14         System.out.print( "\nRollNo: " );
15         int rollNo = sc.nextInt( );
16
17         System.out.print( "\nAddress: " );
18         String address = sc.next( );
19
20         System.out.println( "\nYour Name: " + name + " RollNo: " + rollNo + " Address: " + address );
21         sc.close( );
22     }
23 }
24
```

Console

<terminated> ScannerConsoleInput [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\javaw.exe (Sep 20, 2015, 10:58:33 PM)

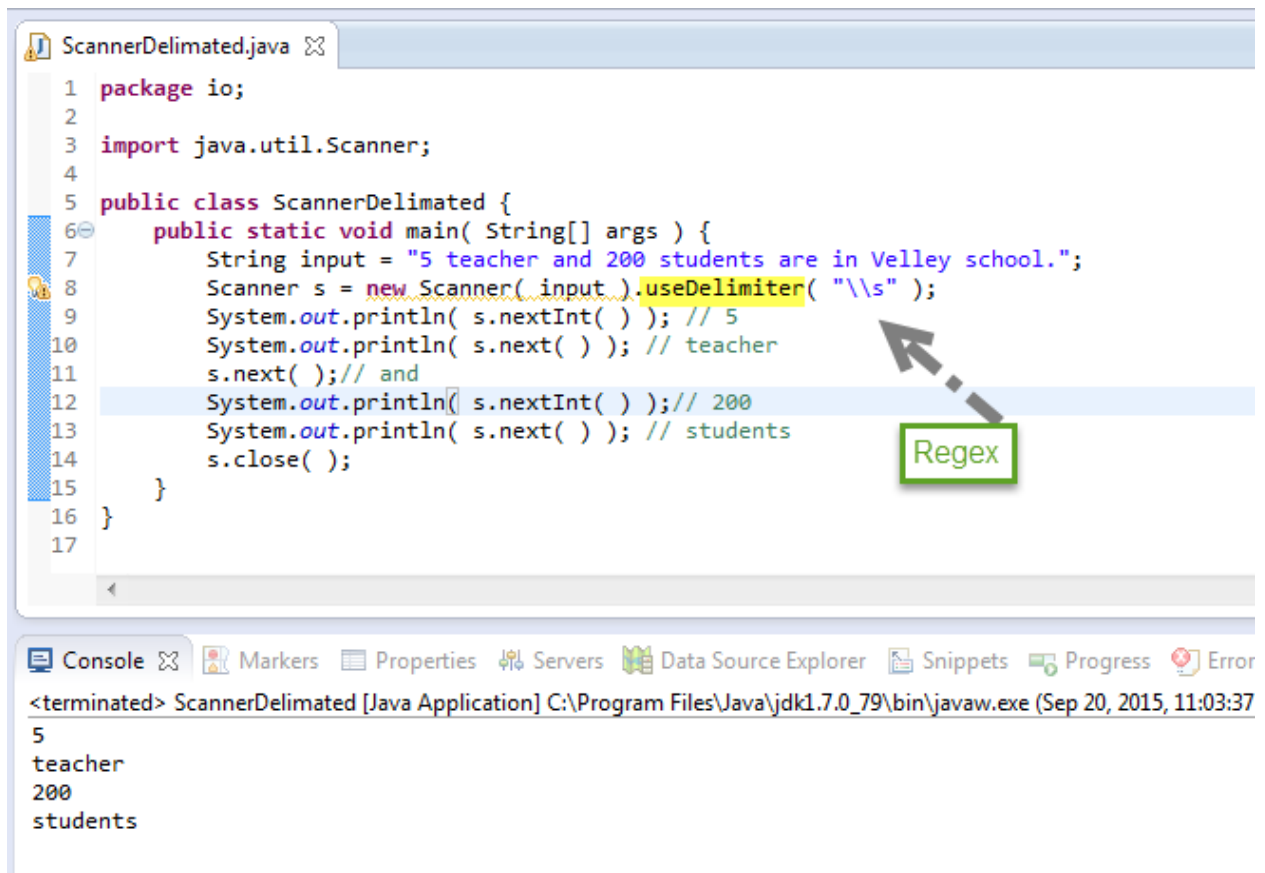
Student Detail.

Name: RAM

RollNo: 11

Address: KTM

Your Name: RAM RollNo: 11 Address: KTM



```
1 package io;
2
3 import java.util.Scanner;
4
5 public class ScannerDelimited {
6     public static void main( String[] args ) {
7         String input = "5 teacher and 200 students are in Velley school.";
8         Scanner s = new Scanner( input ).useDelimiter( "\\s" );
9         System.out.println( s.nextInt( ) ); // 5
10        System.out.println( s.next( ) ); // teacher
11        s.next( ); // and
12        System.out.println( s.nextInt( ) ); // 200
13        System.out.println( s.next( ) ); // students
14        s.close( );
15    }
16 }
17
```

Regex

Console <terminated> ScannerDelimited [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\javaw.exe (Sep 20, 2015, 11:03:37)

```
5
teacher
200
students
```

File Class:

Using the `File` class you can:

- [Check if a file or directory exists.](#)
- [Create a directory if it does not exist.](#)
- [Read the length of a file.](#)
- [Rename or move a file.](#)
- [Delete a file.](#)
- [Check if path is file or directory.](#)
- [Read list of files in a directory.](#)

```

FileExample.java
1 package io;
2
3 import java.io.File;
4
5 public class FileExample {
6
7     @SuppressWarnings( "unused" )
8     public static void main( String[] args ) {
9
10         /* Instantiate File */
11         File file = new File( "D:\\ab.txt" );
12
13         // Check if above path is Directory?
14         boolean isDirectory = file.isDirectory( );
15
16         /* File exist or not */
17         boolean isFileExists = file.exists( );
18
19         /* Create Directory */
20         File fDir = new File( "D:\\temp" );
21         boolean dirCreated = fDir.mkdir( );
22
23         /* List name of all files */
24         String[] fileNames = file.list( );
25
26         /* List of Files */
27         File[] files = file.listFiles( );
28
29     }
30 }
31

```

Serialization/Deserialization in Java

Serialization in java is a mechanism of *writing the state of an object into a byte stream*.

It is mainly used in Hibernate, RMI, JPA, EJB, JMS technologies.

The reverse operation of serialization is called *deserialization*.

The String class and all the wrapper classes implements *java.io.Serializable* interface by default.

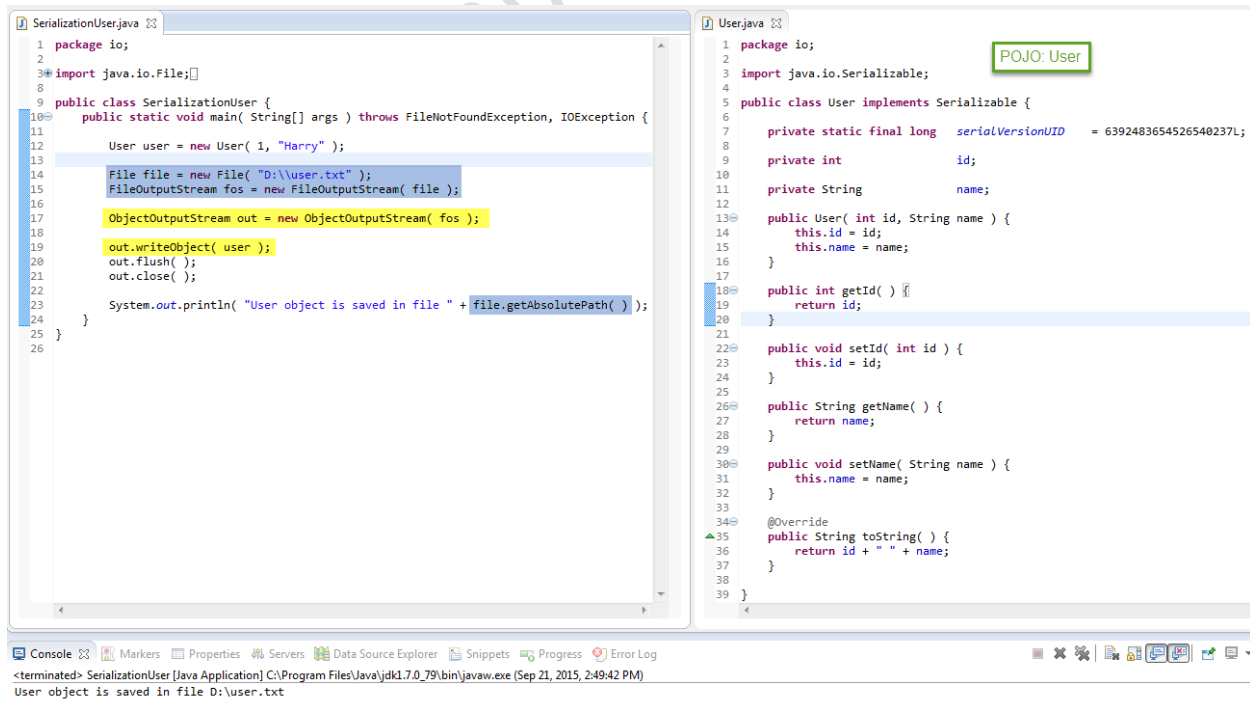
Advantage of Java Serialization

It is mainly used to travel object's state on the network (known as marshaling).

ObjectOutputStream/ObjectInputStream class

The ObjectOutputStream class is used to write primitive data types and Java objects to an OutputStream.

An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.



```
SerializationUser.java
1 package io;
2
3 import java.io.*;
4
5 public class SerializationUser {
6     public static void main( String[] args ) throws FileNotFoundException, IOException {
7         User user = new User( 1, "Harry" );
8         File file = new File( "D:\\user.txt" );
9         FileOutputStream fos = new FileOutputStream( file );
10        ObjectOutputStream out = new ObjectOutputStream( fos );
11        out.writeObject( user );
12        out.flush( );
13        out.close( );
14        System.out.println( "User object is saved in file " + file.getAbsolutePath( ) );
15    }
16 }

User.java
1 package io;
2
3 import java.io.Serializable;
4
5 public class User implements Serializable {
6     private static final long serialVersionUID = 6392483654526540237L;
7     private int id;
8     private String name;
9
10    public User( int id, String name ) {
11        this.id = id;
12        this.name = name;
13    }
14
15    public int getId( ) {
16        return id;
17    }
18
19    public void setId( int id ) {
20        this.id = id;
21    }
22
23    public String getName( ) {
24        return name;
25    }
26
27    public void setName( String name ) {
28        this.name = name;
29    }
30
31    @Override
32    public String toString( ) {
33        return id + " " + name;
34    }
35 }
```

Console: <terminated> SerializationUser [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\javaw.exe (Sep 21, 2015, 2:49:42 PM)
User object is saved in file D:\user.txt

```
DeserializeUser.java
1 package io;
2
3 import java.io.File;
4
5 public class DeserializeUser {
6     public static void main( String[] args ) throws FileNotFoundException, IOException, ClassNotFoundException {
7
8         ObjectInputStream io = new ObjectInputStream( new FileInputStream( new File( "D:\\user.txt" ) ) );
9
10        User user = ( User ) io.readObject( );
11        io.close( );
12
13        System.out.println( user );
14    }
15 }
16
17
18
19
20
21
```

Console | Markers | Properties | Servers | Data Source Explorer | Snippets | Progress | Error Log

<terminated> DeserializeUser [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\javaw.exe (Sep 21, 2015, 2:52:18 PM)

1 Harry

```
public class User implements Serializable {

    private static final long    serialVersionUID    = 6392483654526540237L;

    private transient int        id;
    private String                name;
}
```

id will be not saved in file

Console | <terminated> | 0 Harry