Object Oriented Programming (with JAVA)

by
Dr. Partha Roy,
Associate Professor,
Bhilai Institute of Technology, Durg

UNIT- III

String class. Wrapper classes (Integer, Boolean, Character, etc.). Multi-threading: Thread concept, Thread class, Runnable interface, Creating customized threads, Thread synchronization, Thread class methods. Java I/O: Use of InputStream, OutputStream, Reader and Writer classes for reading from and writing data into disk files.

String class in Java

- •A String is an array of characters terminated by a null character.
- •The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.
- •String literals are constants.
- •The class String includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase, etc.

String class in Java

- The Java language provides special support for the string concatenation operator (+ and +=), and for conversion of other objects to strings.
- String conversions are implemented through the method toString(), defined by Object and inherited by all classes in Java.
- A String object is immutable, so any member methods called upon a string object does not change the calling object but instead creates a new string object as a result.

String class in Java

- •The operator == cannot be used to compare two string objects as the == operator only checks the address to which both the objects are pointing to and not their contents, if the addresses are equal then it gives true otherwise it gives false. So if we have two objects String s1=new String(“abc”); String s2=new String(“abc”); then s1==s2 would give a false value as both refer to different addresses. But s1.equals(s2) would give a true result as equals() method is overriden in String class to compare the contents of the string objects.

String class in Java

Exploring String class member methods

Wrapper Classes

- Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects.
- To wrap primitives into object form, so that we can use primitives as object types as some classes cannot work with primitive data types.
- To use utility method for converting primitive types into String.

  Ex: `String s=Integer.toString(10);`

  //converting primitive 10 into String object.
- To use utility method for converting String containing primitive values into specific prmitive type.

  Ex: `int n=Integer.parseint("100");`

  //converting String 100 into primitive type int 100

Wrapper Classes

Primitive Data Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Wrapper classes

- To create a wrapper object, we use the wrapper class instead of the primitive type. To get the value, we can just print the object:

```
public class Main {
  public static void main(String[] args) {
    Integer myInt = 5;
    Double myDouble = 5.99;
    Character myChar = 'A';
    System.out.println(myInt); //5
    System.out.println(myDouble); //5.99
    System.out.println(myChar); //A
  }
}
```

Wrapper classes

- The following methods are used to get the value associated with the corresponding wrapper object: intValue(), byteValue(), shortValue(), longValue(), floatValue(), doubleValue(), charValue(), booleanValue().

```
public class Main {
  public static void main(String[] args) {
    Integer myInt = 5;
    Double myDouble = 5.99;
    Character myChar = 'A';
    System.out.println(myInt.intValue()); //5
    System.out.println(myDouble.doubleValue()); //5.99
    System.out.println(myChar.charValue()); //A
  }
}
```

Wrapper classes

- Another useful method is the toString() method, which is used to convert wrapper objects to strings.
- In the following example, we convert an Integer to a String, and use the length() method of the String class to output the length of the "string":

```
public class Main {
  public static void main(String[] args) {
    Integer myInt = 100;
    String myString = myInt.toString();
    System.out.println(myString.length()); //3
  }
}
```

Wrapper classes

- Another useful set of methods are “parse” methods that can be used to convert a String containing a primitive data in double quotes into primitive data.

```
public class NewClass{
    public static void main(String[] args){
        String s1="100";
        String s2="123.456";
        int n=Integer.parseInt(s1);
        float f=Float.parseFloat(s2);
        double d=Double.parseDouble(s2);
        System.out.println(n);//100
        System.out.println(f);//123.456
        System.out.println(d);//123.456
    }
}
```

Multithreading

- Threads allows a program to operate more efficiently by doing multiple things at the same time.
- Threads can be used to perform complicated tasks in the background without interrupting the main program.
- Each thread has its own private stack and registers, including program counter. These are essentially the things that threads need in order to be independent.

Multithreading

- •Java threads are objects like any other Java objects. Threads are instances of class **java.lang.Thread**, or instances of subclasses of this class.
- In addition to being objects, java threads can also execute code.
- •Creating a thread in Java is done like this:

```
Thread th = new Thread();
```

- •To start the thread we will call its start() method, like this:

```
th.start();
```

Multithreading

- The **start()** method calls the **run()** method which should contain the code the thread should execute.
- Threads execute the code inside the run() method.
- Prototype declarations of start() and run() methods:

```
public void start();
public void run();
```

- There are two ways to specify what code the thread should execute.

1.*The first is to create a* ***subclass of Thread*** *and override the* ***run()*** *method.*

2.*The second method is to pass an object that* ***implements Runnable interface*** *to the* ***Thread constructor****.*

Creating a Thread

There are two ways to create a thread.

1) It can be created by extending the **Thread class** and overriding its run() method:

```
public class Main extends Thread {
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

2) Another way to create a thread is to implement the **Runnable interface**:

```
public class Main implements Runnable {
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

Running Threads

- If the class **extends the Thread class**, the thread can be run by creating an instance of the class and call its start() method:

```
public class Main extends Thread {
  public static void main(String[] args) {
    Main th = new Main();
    th.start();
    System.out.println("This code is outside of the
thread");
  }
  public void run() {
    System.out.println("This code is running in a
thread");
  }
}
```

📣 ***Run this program multiple times and note your observation***

Running Threads

- If the class **implements the Runnable interface**, the thread can be run by passing an instance of the class to a Thread object's constructor and then calling the thread's start() method:

```
public class Main implements Runnable {
  public static void main(String[] args) {
    Main obj = new Main();
    Thread th = new Thread(obj);
    th.start();
    System.out.println("This code is outside of the thread");
  }
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

📣 ***Run this program multiple times and note your observation***

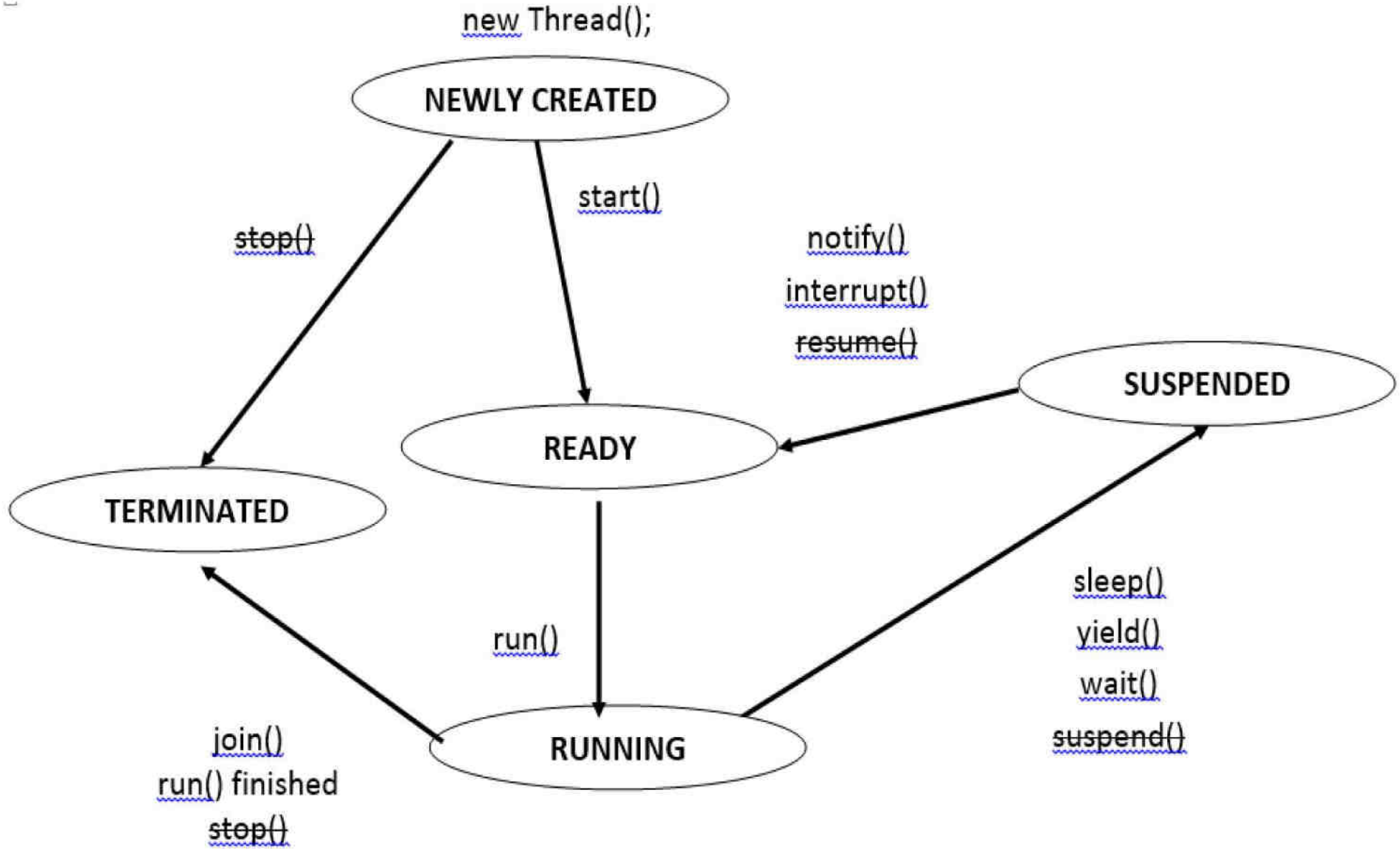
What happens when we call the run() as well as the start() ?

```
class MyThread extends Thread{
    public void run(){
        long tid=Thread.currentThread().getId();
        System.out.println("Thread Id of run(): "+tid);
        System.out.println("run() method ends here");
    }
}

public class MyThreads{
    public static void main( String args [] ){
        MyThread ob=new MyThread();
        ob.start();
        ob.run();
        long tid=Thread.currentThread().getId();
        System.out.println("Thread Id of main(): "+tid);
        System.out.println("main() method ends here ");
    }
}
```

📣 ***Run this program multiple times and note your observation***

Thread Lifecycle



Pre-empting threads

- •Putting a thread into suspended or ready state from running state is called pre-empting a thread.
- •When a thread is pre-empted the thread scheduler puts the thread into ready after the completion of thread pre-emption.
- •There are three methods using which pre-empting of threads is possible:
  - ***sleep()***
  - ***yield()***
  - ***join()***

The sleep() method in Thread class

- **Thread.sleep()** causes the current thread to suspend execution for a specified period.
- This is an efficient means of making processor time available to the other threads of an application or other applications that might be running on a computer system.
- The **java.lang.Thread.sleep(long millis)** method causes the currently executing thread to sleep for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.
- After the lapse of the specified milliseconds the thread scheduler puts the thread into ready state.
- Method prototype:

```
public static void sleep(long millis) throws InterruptedException
```

- ***The sleep() method does not compel the thread to leave a synchronized code (monitor).***

The sleep() method in Thread class

Code-1 to demonstrate the use of sleep() method.

Code-2 to demonstrate the use of sleep() method.

The yield() method of Thread

- The **java.lang.Thread.yield()** method causes the currently executing thread object to temporarily pause and allow other threads to execute.
- The thread scheduler puts the running thread into ready state.
- This method can be one solution to provide pre-emptive-ness in non-pre-emptive systems.
- The thread that wants other threads to continue in between of its own execution would call the yield() method occasionally.
- Following is the declaration for java.lang.Thread.yield() method:

```
public static void yield()
```

- The scheduler can re-schedule the same thread again after yield().
- ***The yield() method does not compel the thread to leave a synchronized code (monitor).***

The yield() method of Thread

Code to demonstrate the use of yield() method.

The "join()" method of Thread class

- •When a thread T1 wants to wait until another thread T2 completes, then T1 in its own code would call T2.join(), this would cause T1 to wait until T2 completes fully.
- •When T1 thread calls T2.join() then T1 would immediately go to waiting state (leaving the running state) until T2 completes fully.
- •Following is the declaration for join() method:

```
public final void join() throws InterruptedException
```

- •Input Parameters: void
- •Return Value: This method does not return any value.
- •Exception: "InterruptedException" -- if any thread has interrupted the current thread. The interrupted status of the current thread is cleared when this exception is thrown. It is a checked exception and so the method has to be enclosed within try-catch block.

The "join()" method of Thread class

The join() method can give rise to two main problems:

- **1. Deadlock:** When a thread T1 calls join() on another thread T2 and T2 thread calls join() on T1 thread. To solve this join(long millis) can be used, so waiting thread can resume after given milliseconds in the input argument completes.
- **2. Starvation:** When T1 thread calls join() on T2 thread and the T2 thread goes into infinite loop and never ends. To solve this join(long millis) can be used, so waiting thread can resume after given milliseconds in the input argument completes.

The "join()" method of Thread class

Example 1: when two threads join.

Example 2: making the parent thread join inside the child thread.

Example 3: illustrating DEADLOCK.

Example 4: illustrating STARTVATION of parent thread.

The "isAlive()" method of Thread class

- •Description: The ***java.lang.Thread.isAlive()*** method tests if this thread is alive. A thread is alive if it has been started and has not yet finished.
- •Declaration

```
public final boolean isAlive()
```

- •Parameters: void
- •Return Value: This method returns true if this thread is alive, false otherwise.

Example: demonstrating isAlive() and join() methods.

Interrupting a Thread

- •An interrupt is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate.
- •The programmer has to decide that after it gets If(Thread.interrupted()==true) then what should be done, either the return statement can be used to exit from the current method or do something else.
- •The 3 methods provided by the Thread class for thread interruption:

```
public void interrupt()
public static boolean interrupted()
public boolean isInterrupted()
```

Interrupting a Thread

- If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing ***InterruptedException***.
- If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true.
- The method isInterrupted() is used to check whether a thread is in running state or suspended state. This method has no input arguments and returns a boolean value true if the thread object is interrupted otherwise returns false if not interrupted. The interruption can be caused by any event that puts the thread to suspended state.
- The interrupt() method does not compel the thread to leave a synchronized code (monitor).

Interrupting a Thread

Example 1: using sleep() and interrupt().

Example 2: using infinite sleep() and interrupt()

Synchronization between threads

- •When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen results due to concurrency issue. For example if multiple threads try to write within a same file then they may corrupt the data because one of the threads can overwrite data or while one thread is opening the same file at the same time another thread might be closing the same file.
- •So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called monitors.
- •Each object in Java is associated with a monitor, which a thread can lock or unlock.
- •Only one thread at a time may hold a lock on a monitor.

Synchronization between threads

- •Java programming language provides a very handy way of creating threads and synchronizing their task by using synchronized blocks.
- •We can keep shared resources within the synchronized block so that only one thread at a time can access the shared resource.
- •There are two types of locks available for threads:
  1. **Object level lock.**
  2. **Class level lock.**

Object level lock

- *•When synchronized keyword is used along with method declaration or synchronized block has the reference of an object.*
- *<access-mode> <return-type> synchronized method-name(i/p arg. list) { … statements in method … }*

Ex:
```
public void synchronized test() {
System.out.println("TEST"); }
```

- *synchronized(reference) { … statements in block … }*

Ex:
```
synchronized(this) {
System.out.println("TEST"); }
```

Class level lock

- ***When synchronized keyword is used along with static method declaration or synchronized block has the class object of the class-name.***
- ***<access-mode> <return-type> static synchronized method-name(i/p arg. list) { ….}***

Ex:
```
public void static synchronized test() {
System.out.println("TEST"); }
```

- ***synchronized(class-name.class)***
  { … statements in block … }

Ex:
```
synchronized(A.class) {
System.out.println("TEST"); }
```

Synchronization between threads

- *A thread can acquire multiple locks on multiple objects but multiple threads cannot acquire lock for same object.*

Example 1: synchronized method – object level lock.

Example 2: synchronized block – object level lock.

Example 3: static synchronized method – class level lock.

Example 4: synchronized block – class level lock.

Multithreading Practical Scenarios

Example 5: Accessing a shared object by threads without synchronization.

Example 6: Accessing a shared object by threads with synchronization.

Inter-thread (inter-process) communication

- •Thread expecting updation of an object should call **wait()** method on that object.
- •Thread performing updation on that object should call **notify()** or **notifyAll()** method on that object.
- •Then the waiting thread would get that notification and can use the updated value of the object.
- •**wait(), notify()** and **notifyAll()** methods are present in ***Object class not in Thread class***.

Inter-thread (inter-process) communication

- •When a thread is calling wait(), notify() or notifyAll() methods on an object then the thread should be the owner of that object that means the thread should get attain the lock of that object (object level lock), otherwise **IllegalMonitorStateException** will be generated in runtime.
- •So **wait(), notify() or notifyAll()** methods can only be called from **synchronized** area.
- •When a thread acquires lock of an object and invokes the **wait()** method on that object then that thread immediately releases that lock and enters waiting state.
- •The thread calling notify()/notifyAll() method on an object, releases the lock on that object subsequently, not necessarily immediately.

Inter-thread (inter-process) communication

- **wait( )** tells the calling thread to give up the monitor 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.

Inter-thread (inter-process) communication

- **final void wait( ) throws InterruptedException**
- **final void notify( )**
- **final void notifyAll( )**
- ***The wait() method compels the thread to leave a synchronized code (monitor).***

Inter-thread (inter-process) communication

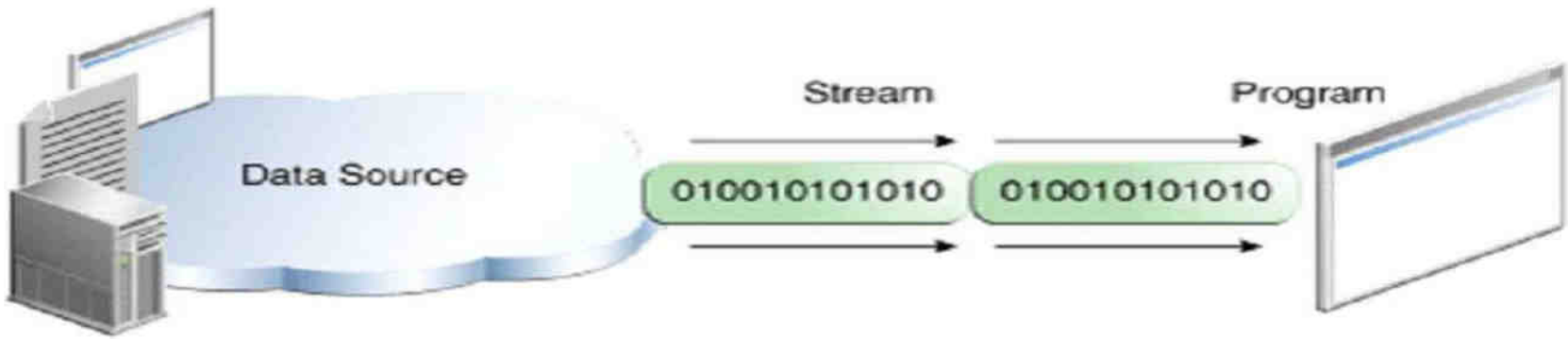
- Example-1 without wait() and notify().
- Example-2 with wait() and notify().

Java Input Output Classes

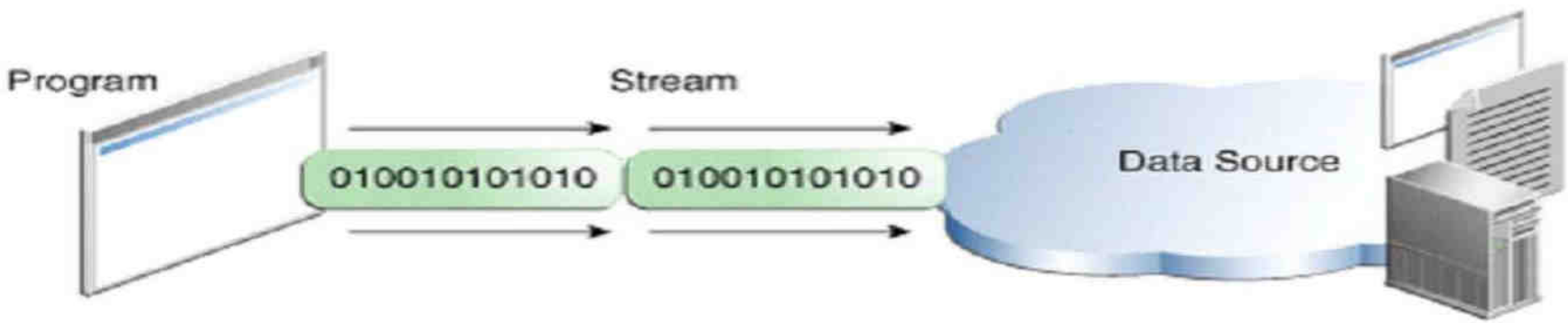
- •A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.
- •Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects.
- •Some streams simply pass on data; others manipulate and transform the data in useful ways.
- •No matter how they work internally, all streams present the same simple model to programs that use them: A stream is a sequence of data.
- •A program uses an input stream to read data from a source, one item at a time.

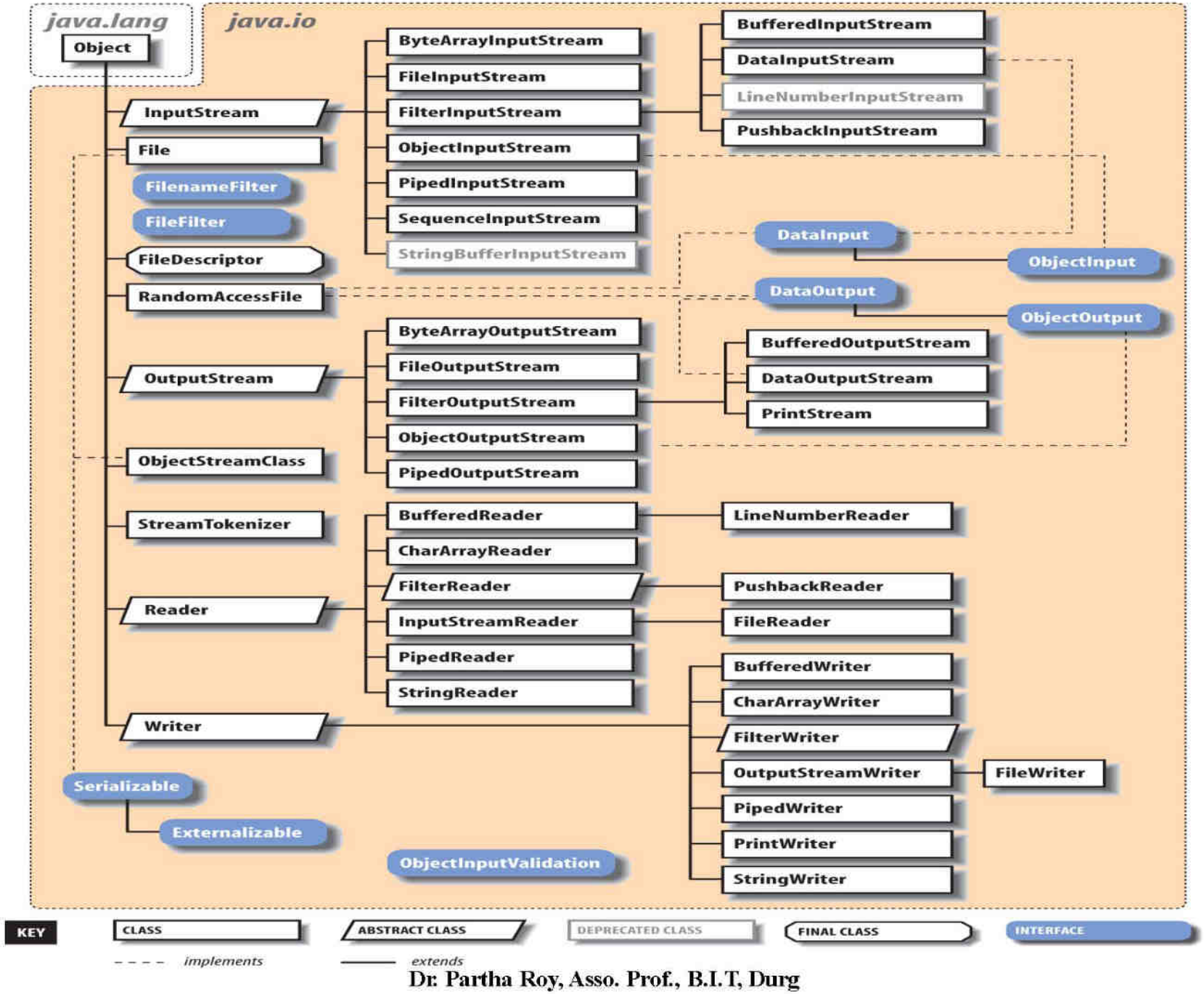
Java Input Output Classes

Reading information into a program.



Writing information from a program.





File class

- Package name: **java.io**
- Java File class represents the files and directory pathnames in an abstract manner.
- File class is used for creation of files and directories, file searching, file deletion, etc.
- While using the File class constructor we need to provide a String that represents the file-name or the path along with file-name.
- The File class constructors creates a new File instance pointing towards the file-name, that does NOT ensure that a new file would be created if it does not exist.

File class

- •The file-name can also contain the detailed path name along with the file-name.
- •If the file-name is not prefixed with the path name then the file is considered to be present in the current working directory.
- •If the file-name exists physically then the new File instance would point towards the file-name otherwise a new file will NOT be created.

File class: Constructors

- ***File(String file-name)***
- A new File class object is created which points towards the physical file present in the system.
- If the file is not present then it only holds the name of the file and awaits creation of actual file in the storage space available.
- ***File(String parent, String child)***
- Creates a new File class instance that points towards the file name denoted by the String child. The String parent denotes the directory in which the file exists.

File class: Member methods

- ***public String getName()***

Returns the name of the file or directory pointed by the calling File class object.

- ***public String getPath()***

Returns the path string in which the calling File class object is pointing to.

- ***public boolean exists()***

Returns true if the file pointed by the calling File class object physically exists in the system, otherwise returns false.

- ***public boolean isDirectory()***

Returns true if the calling File class object points to a directory or else returns false.

- ***public boolean isFile()***

Returns true if the calling File class object points to a file or else returns false.

File class: Member methods

- ***public long length()***

Returns the size of the file in terms of Bytes, pointed by the calling File class object.

- ***public boolean createNewFile() throws IOException***

Physically creates a new file with the file-name and path-name pointed by the calling File class object.

- ***public boolean delete()***

Physically deletes the file pointed by the calling File class object.

- ***public String[] list()***

Returns an array of String containing the names of the files and directories present in the directory pointed by the calling File class object.

- ***public boolean mkdir()***

Creates a new directory whose name is denoted by the calling File class object.

File class: Member methods

- **Example of File class methods.**

FileInputStream and FileOutputStream

- A **FileInputStream** obtains input bytes from a file in a file system. What files are available depends on the host environment.
- **FileInputStream** is meant for reading streams of raw bytes.
- A **FileOutputStream** is an output stream for writing data to a File. Whether or not a file is available or may be created depends upon the underlying platform.
- Some platforms, in particular, allow a file to be opened for writing by only one **FileOutputStream** (or other file-writing object) at a time. In such situations the constructors in this class will fail if the file involved is already open.

FileInputStream and FileOutputStream

- ***All the Constructors throw FileNotFoundException if the file object or string refers to invalid file or location.***
- **FileInputStream(File file)**

Creates a FileInputStream by opening a connection to an actual file, the file named by the File object file in the file system.

- **FileInputStream(String name)**

Creates a FileInputStream by opening a connection to an actual file, the file named by the path name name in the file system.

- **FileOutputStream(File file)**

Creates a file output stream to write to the file represented by the specified File object.

FileInputStream and FileOutputStream

- **FileOutputStream(File file, boolean append)**

Creates a file output stream to write to the file represented by the specified File object.

- **FileOutputStream(String name)**

Creates a file output stream to write to the file with the specified name.

- **FileOutputStream(String name, boolean append)**

Creates a file output stream to write to the file with the specified name.

FileInputStream and FileOutputStream Example

FileWriter class

- FileWriter class (**java.io.FileWriter**) is used to write character-oriented data to a file.
- It is character-oriented class which is used for file handling in java.
- Unlike FileOutputStream class, we don't need to convert string into byte array because it provides method to write string directly.
- ***public class FileWriter extends OutputStreamWriter***

FileWriter class

Constructors of FileWriter class:

- **FileWriter(String file)**

Creates a new file. It gets file name in string.

- **FileWriter(String file, boolean append)**

Creates a new file with specified file name as string.

- **FileWriter(File file)**

Creates a new file. It gets file name in File object.

- **FileWriter(File file, boolean append)**

Creates a new file. It gets file name in File object.

FileWriter class

Methods of FileWriter class:

- **void write(String text)**

It is used to write the string into FileWriter.

- **void write(char c)**

It is used to write the char into FileWriter.

- **void write(char[] c)**

It is used to write char array into FileWriter.

- **void flush()**

It is used to flush(clean buffer) the data of FileWriter.

- **void close()**

It is used to close the FileWriter.

FileWriter example

```
import java.io.FileWriter;

public class NewClass {
    public static void main(String[] arg){
        try{
            FileWriter fw=new
             FileWriter("D:\\MyJavaProgs\\Test.txt",true);
            fw.write("Welcome to FileWriter");
            fw.close();
        }catch (Exception e){}
    }
}
```

FileReader class

- FileReader class (**java.io.FileReader**) is used to read data from the file.
- It returns data in byte format like FileInputStream class.
- It is character-oriented class which is used for file handling in java.

public class FileReader extends InputStreamReader

FileReader class

Constructors of FileReader class:

- **FileReader(String file)**

It gets filename in string. It opens the given file in read mode. If file doesn't exist, it throws ***FileNotFoundException***.

- **FileReader(File file)**

It gets filename in file instance. It opens the given file in read mode. If file doesn't exist, it throws ***FileNotFoundException***.

FileReader class

Methods of FileReader class:

- **int read():**

It is used to return a character in ASCII form. It returns -1 at the end of file.

- **void close():**

It is used to close the FileReader class.

FileReader class example

```
import java.io.FileReader;

public class NewClass {
    public static void main(String[] arg){
        try{
            FileReader fr=new
                FileReader("D:\\MyJavaProgs\\Test.txt");
            int n;
            while((n=fr.read())!=-1){
            System.out.print((char)n);
        }
            fr.close();
        }catch (Exception e){}
    }
}
```

Object Streams and Serialization

- •Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

Serialization:

- •Converting object from user supported form to transport supported form.
- •Converting an object in java supported form to disk-file or network supported form.

De-Serialization:

- •Converting an object in transport supported form to user supported form.
- •Converting an object in disk-file or network supported form to java supported form.

Object Streams and Serialization

- •After a serialized object has been written into a file, it can be read from the file and deserialized, i.e. the type information and bytes that represent the object and its data can be used to recreate the object in memory.
- •Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.
- •Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.

Object Streams and Serialization

- The **ObjectOutputStream** class contains many write methods for writing various data types, but one method in particular stands out:

public final void writeObject(Object x) throws IOException

The above method serializes an Object and sends it to the output stream.

- Similarly, the **ObjectInputStream** class contains the following method for deserializing an object:

public final Object readObject() throws IOException, ClassNotFoundException

This method retrieves the next Object out of the stream and deserializes it. The return value is Object, so you will need to cast it to its appropriate data type. Throws **EOFException** if EOF is reached.

Object Streams and Serialization

- To make a Java object serializable we implement the **java.io.Serializable** interface. This is only a marker interface which tells the Java platform that the object is serializable.
- Making a class Serializable in Java is very easy, our Java class just needs to use implements **java.io.Serializable** interface and JVM will take care of serializing object in default format.
- Serializable interface exists in java.io package and forms core of java serialization mechanism. It doesn't have any method and also called **Marker Interface** in Java.

ObjectInputStream and ObjectOutputStream constructors

- **ObjectInputStream()**

Provide a way for subclasses that are completely reimplementing ObjectInputStream to not have to allocate private data just used by this implementation of ObjectInputStream.

- **ObjectInputStream(InputStream in)**

Creates an ObjectInputStream that reads from the specified InputStream.

- **ObjectOutputStream()**

Provide a way for subclasses that are completely reimplementing ObjectOutputStream to not have to allocate private data just used by this implementation of ObjectOutputStream.

- **ObjectOutputStream(OutputStream out)**

Creates an ObjectOutputStream that writes to the specified OutputStream.

Serialization and De-Serialization basic Example

```
import java.io.*;
class C1  implements Serializable
{
    int i=10;
    int j=20;
}
public class NewClass10
{
    public static void main(String ar[]) throws Exception
    {
        ObjectOutputStream oos=new ObjectOutputStream(new FileOutputStream("D:\\SerFile.txt"));
        C1 ob1=new C1();
        ob1.i=100;
        ob1.j=100;
        System.out.println("Before Serialization:");
        System.out.println(ob1.i+"......"+ob1.j);
        oos.writeObject(ob1);
        oos.close();
        ObjectInputStream ois=new ObjectInputStream(new FileInputStream("D:\\SerFile.txt"));
        C1 ob2=(C1)ois.readObject();
        System.out.println("After De-Serialization:");
        System.out.println(ob2.i+"......"+ob2.j);
    }
}
```

Output:

Before Serialization:

100.....100

After De-Serialization:

100.....100

InputStreamReader class

- An InputStreamReader is a bridge from byte streams to character streams.
- It reads bytes and decodes them into characters using a specified charset.
- The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted.
- Each invocation of one of an InputStreamReader's read() methods may cause one or more bytes to be read from the underlying byte-input stream.

OutputStreamWriter class

- An OutputStreamWriter is a bridge from character streams to byte streams.
- Characters written to it are encoded into bytes using a specified charset.
- The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted.
- Each invocation of a write() method causes the encoding converter to be invoked on the given character(s).
- The resulting bytes are accumulated in a buffer before being written to the underlying output stream.
- The size of this buffer may be specified, but by default it is large enough for most purposes.

InputStreamReader & OutputStreamWriter constructors

- **InputStreamReader(InputStream in)**

Creates an InputStreamReader that uses the default charset.

- **OutputStreamWriter(OutputStream out)**

Creates an OutputStreamWriter that uses the default character encoding.

InputStreamReader and OutputStreamWriter Example

BufferedReader and BufferedWriter

- **BufferedWriter** writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings.
- **BufferedReader** reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.
- The buffer size may be specified, or the default size may be used. The default is large enough for most purposes.

BufferedReader and BufferedWriter

Constructors:

- **public BufferedWriter(Writer out)**

Creates a buffered character-output stream that uses a default-sized output buffer.

- **public BufferedReader(Reader in)**

Creates a buffering character-input stream that uses a default-sized input buffer.

BufferedReader and BufferedWriter example

End of UNIT-III