```
}
}
public void innerCreate()
{
Inner in=new Inner();
in.innerFn();
}
public void outerFn()
{
System.oaut.printIn("I is "+ i);
}
public static void main(String s[ ] )
{
Out out=new Outer();
out.innerCreate();
}
}
```

ADDING CLASSES TO EXISTING PACKAGES

Suppose a package A contains a public class, Class A, and we want to add another public class, Class B to this package.

```
package packageA;
public class ClassA;
{
//(body of classA)
}
```

```
package packageB;
public class ClassB
{
//(body of ClassB)
}
```

Store the source and compiled files ClassB.java and ClassB.class in the directory **packageA**. we can add non-public classes also in this manner. When the package, **packageA**, is imported, both the classes **ClassA** and **ClassB** are imported, as they are contained in that package.

A **.java** file can have only one public class. So, if we want to create a package with multiple public classes in it, create the classes in separate source files and declare the package statement

package packagename;

at the top of each source file. Switch to the subdirectory created with the package name, and compile each source file. Now, the package contains **.class** files of all the source files.

INTERTHREAD COMMUNICATION

To avoid wastage of precious time of CPU, or to avoid polling, Java includes an interprocess communication mechanism via the wait( ), notify( ), and notifyAll( ) methods. These methods are implemented as final methods, so all classes have them. These three methods can be called only from within a synchronized method.

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( ) or notifyAll().
notify( ): Wakes up a single thread that is waiting on this object's monitor.
notifyAll( ): Wakes up all threads that are waiting on this object's monitor, the highest priority Thread will be run first.

These methods are declared within objects as following:

```
final void wait( ) throws Interrupted Exception
final void notify( )
final void notifyAll( )
```

These methods enable you to place threads in a waiting pool until resources become available to satisfy the Thread's request. A separate resource monitor or controller process can be used to notify waiting threads that they are able to execute.

//program

```
class WaitNotifyTest implements Runnable
{
WaitNotifyTest ()
{
Thread th = new Thread (this);
th.start();
}
synchronized void notifyThat ()
{
System.out.println ("Notify the threads waiting");
this.notify();
}
synchronized public void run()
{
try
{
System.out.println("Thead is waiting....");
this.wait ();
}
catch (InterruptedException e){}
System.out.println ("Waiting thread notified");
}
}
class runWaitNotify
{
public static void main (String args[])
{
WaitNotifyTest wait_not = new WaitNotifyTest();
Thread.yield ();
wait_not.notifyThat();
}
}
```

I/O in java

Java input and output are based on the use of streams, or sequences of bytes that travel from a source to a destination over a communication path. If a program is writing to a stream, you can consider it as a stream's source. If it is reading from a stream, it is the stream's destination. The communication path is dependent on the type of I/O being performed. It can consist of memory-to-memory transfers, a file system, a network, and other forms of I/O. Streams are powerful because they abstract the details of the communication path from input and output operations. This allows all I/O to be performed using a common set of methods. These methods can be extended to provide higher-level custom I/O capabilities.

Three streams given below are created automatically:
System.out - standard output stream
System.in - standard input stream
System.err - standard error.

An **InputStream** represents a stream of data from which data can be read. Again, this stream will be either directly connected to a device or else to another stream.

An **OutputStream** represents a stream to which data can be written. Typically, this stream will either be directly connected to a device, such as a file or a network connection, or to another output stream.

STREAMS AND STREAM CLASSES

There are two types of streams: byte streams, and character streams.

Byte streams carry integers with values that range from 0 to 255. A diversified data can be expressed in byte format, including numerical data, executable programs, and byte codes – the class file that runs a Java program.

Character Streams are specialised types of byte streams that can handle only textual data.

Byte Stream Classes

Java defines two major classes of byte streams: InputStream and OutputStream. To provide a variety of I/O capabilities subclasses are derived from these InputStream and OutputStream classes.

InputStream class

The InputStream class defines methods for reading bytes or arrays of bytes, marking locations in the stream, skipping bytes of input, finding out the number of bytes available for reading, and resetting the current position within the stream. An input stream is automatically opened when created. The close() method can explicitly close a stream.

Methods of InputStream class

The basic method for getting data from any InputStream object is the read() method.

public abstract int read() throws IOException: This reads a single byte from the input stream and returns it to the stream.

public int read(byte[] bytes) throws IOException: fills an array with bytes read from the stream and returns the number of bytes read.

public int read(byte[] bytes, int offset, int length) throws IOException: fills an array from a stream starting at position offset, up to the length of the bytes. This returns either the number of bytes read or 1 for end of file.

public int available() throws IOException: the read()method always blocks when there is no data available. To avoid blocking, the program might need to ask ahead of time exactly how many bytes it can safely read without blocking. This method returns this number.

public long skip(long n): the skip() method skips over n bytes (passed as argument of skip()method) in a stream.

OutputStream class

The OutputStream defines methods for writing bytes or arrays of bytes to the stream. An output stream is automatically opened when created. An Output stream can be explicitly closed with the close() method.

Methods of OutputStream class

public abstract void write(int b) throws IOException: writes a single byte of data to an output stream.
public void write(byte[] bytes) throws IOException: writes the entire contents of the bytes array to the output stream.
public void write(byte[] bytes, int offset, int length) throws IOException: writes length, number of bytes starting at position offset from the bytes array.

The Java.io package contains several subclasses of InputStream and OutputStream that implement specific input or output functions. Some of these classes are:

FileInputStream and FileOutputStream: read data from or write data to a file on the native file system.

Character Stream Classes

Character Streams are defined by using two class **Java.io.Reader and Java.io.Writer** hierarchies.

Both Reader and Writer are the abstract parent classes for character stream based classes in the Java.io package. Reader classes are used to read 16-bit character streams and Writer classes are used to write to 16-bit character streams. The methods for reading from and writing to streams found in these two classes and their descendant classes (which we will discuss in the next section of this unit) given below:

```
int read()
int read(char cbuf[])
int read(char cbuf[], int offset, int length)
int write(int c)
int write(char cbuf[])
int write(char cbuf[], int offset, int length)
```

Reading Console Input

Java takes input from the console by reading from System.in. It can be associated with these sources with reader and sink objects by wrapping them in a reader object. For System.in an InputStreamReader is appropriate. This can be further wrapped in a BufferedReader as given below, if a line-based input is required.

```
BufferedReader br = new BufferedReader (new InputStreamReader(System.in))
```

After this statement br is a character-based stream that is linked to the console through Sytem.in

The program given below is used for receiving console input in any of your Java applications.

//program

```
import Java.io.*;
class ConsoleInput
{
static String readLine()
{
StringBuffer response = new StringBuffer();
try
{
BufferedInputStream buff = new BufferedInputStream(System.in);
int in = 0;
char inChar;
do
{
in = buff.read();
inChar = (char) in;
if (in != -1)
{
response.append(inChar);
}
} while ((in != 1) & (inChar != '\n'));
buff.close();
return response.toString();

}
catch (IOException e)
{
System.out.println("Exception: " + e.getMessage());
return null;
}
}
public static void main(String[] arguments)
{
System.out.print("\nWhat is your name?");
String input = ConsoleInput.readLine();
System.out.println("\nHello, " + input);
}
}
```

Writing Console Output

we have to use System.out for standard Output in Java. It is mostly used for tracing errors, or for sample programs. These sources can be associated with a writer and sink objects by wrapping them in writer object. For standard output, an OutputStreamWriter object can be used, but this is often used to retain the functionality of print and println methods. In this case, the appropriate writer is PrintWriter. The second argument to PrintWriter constructor requests that the output will be flushed whenever the println method is used. This avoids the need to write explicit calls to the flush method in order to cause the pending output to appear on the screen. PrintWriter is different from other input/output classes as it doesn't throw an IOException. It is necessary to send check Error message, which returns true if an error has occurred. One more side effect of this method is that it flushes the stream.

```
PrintWriter pw = new PrintWriter (System.out, true)
```

Object serialization

Serialization takes all the data attributes, writes them out as an object, and reads them back in as an object. For an object to be saved to a disk file it needs to be converted to a serial form. An object can be used with streams by implementing the serializable interface. The serialization is used to indicate that objects of that class can be saved and retrieved in serial form. Object serialization is quite useful when you need to use object persistence. By object persistence, the stored object continues to serve the purpose even when no Java program is running, and stored information can be retrieved in a program so it can resume functioning, unlike the other objects that cease to exist when object stops running.