Chapter 14: Exception Handling

14.1 What are Exceptions?

- **Exceptions in java** are any abnormal, unexpected events or extraordinary conditions that may occur at runtime
- They could be file not found exception, unable to get connection exception and so on.
- On such conditions java throws an exception object.
- Java Exceptions are basically Java objects
- Java exception handling is used to handle error conditions in a program systematically by taking the necessary action
- Any exceptions not specifically handled within a Java program are caught by the Java run time environment
- Java exceptions are raised with the **throw** keyword and handled within a **catch** block.

Find output of the program

```
class ExceptionExample1{
    public static void main(String[] args){
        System.out.println("Result: "+3/0);
    }
}
```

- There was **no compilation error**, but since we were trying to divide 3 by 0(zero), it is **run time error (exception)** and you will see the following error message on the console:
- Exception in thread "main" java.lang.ArithmeticException: / by zero at ExceptionExample1.main(ExceptionExample1.java:3)

Find output of the program

```
class ExceptionExample2{
   public static void main(String[] args){
     String s1;
     System.out.println("Result: "+s1.length());
   }
}
```

- This time we have **a compilation error**, since compiler recognize that the object s1 is not initialized.
- All local variables/object references must be initialized before use as they do not contain default value like instance members.

Find output of the program

```
class ExceptionExample3{
  public static void main(String[] args){
    String s1=null;
    System.out.println("Result: "+s1.length());
  }
}
```

- This time we got no compilation error.
- Run time error (Exception) is generated.
- Exception in thread "main" java.lang.NullPointerException at ExceptionExample3.main(ExceptionExample3.java:5)

- So exceptions (like NullPointerException, ArithmeticException) are classes in java.lang package

14.2 Exception Classes Hierarchy

Diagram below provides a glimpse of exception class hierarchy. At the top of the hierarchy is Throwable class, which defines the common characteristics of exception classes.

An exception is a subclass of the Exception/Error class, both of which are subclasses of the Throwable class.

The Throwable class provides a String variable that can be set by the subclasses to provide a detail message that provides more information of the exception occurred

All classes of Throwables define a one-parameter constructor that takes a string as the detail message

The class Throwable provides getMessage() function to retrieve an exception

14.3 Types of Exceptions

- The class Exception represents exceptions that a program faces due to abnormal or special conditions during execution.
- Exceptions can be of 2 types: Checked (Compile time Exceptions)/ Unchecked (Run time Exceptions).

14.4 Unchecked Exceptions

- Unchecked exceptions are RuntimeException and any of its subclasses
- ArrayIndexOutOfBounds, NullPointerException and so on are all subclasses of the java.lang.RuntimeException class, which is a subclass of the Exception class.
- These are basically business logic programming errors

14.5 Handling Exceptions

try

{
<code>

} catch (<exception type> <parameter>) {

// 0 or more <statements>

}

finally {

// finally block <statements>

}

try

- The java code that you think may produce an exception is placed within a try block for a
  suitable catch block to handle the error
- If no exception occurs the execution proceeds with the finally block else it will look for the matching catch block to handle the error.
- Again if the matching catch handler is not found execution proceeds with the finally block and the default exception handler throws an exception.
- If an exception is generated within the try block, the remaining statements in the try block will not be executed

catch

- Exceptions thrown during execution of the try block can be caught and handled in a catch block.
- On exit from a catch block, normal execution continues and the finally block is executed

```
class ExceptionExample5{
  public static void main(String[] args){
     try{
       System.out.println(3/0);
       System.out.println(“In try");
     }
     catch(ArithmeticException e){
       System.out.println("Exception: "+e.getMessage());
     }
     System.out.println("Hello");
  }
}
```

- On the exit from catch block, statements written after catch blocks will be executed

finally

- A finally block is always executed, regardless of the cause of exit from the try block, or whether any catch block was executed.
- Generally finally block is used for freeing resources, cleaning up, closing connections etc.

Remember

- For each try block there can be zero or more catch blocks, but only one finally block
- The catch blocks and finally block must always appear in conjunction with a try block
- A try block must be followed by either at least one catch block or one finally block.
- The order exception handlers in the catch block must be from the most specific exception

Find output

```
class ExceptionExample{
  public static void main(String[] args){
     String s1=null;
     try{
                System.out.println(3/1);
                System.out.println(s1.length());
     } catch(ArithmeticException e){
                System.out.println("Exception: "+e.getMessage());
     }
     finally    {
       System.out.println("Finally");
     }
     System.out.println("Last line");
  }
}
```

Output:
3
Finally
Exception in thread “main” java.lang.NullPointerException at
ExceptionExample.main(ExceptionExample.java:5)

14.6 The keyword throw

- A program can explicitly throw an exception using the throw statement besides the implicit exception thrown.
- Syntax:
    - throw <throwableInstance>;
- The Exception reference must be of type Throwable class or one of its subclasses
- A detail message can be passed to the constructor when the exception object is created.
- The use of keyword throw is to throw an exception object. Below is the example illustrating the use of throw:

```
class ExceptionExample7{
  public static void main(String[] args){
       int x=10;
       int y=0;
       if(y==0)
          {
             throw new ArithmeticException("Invalid attempt of division");
          }
          System.out.println("Result: "+x/y);
    }
}
```

- Since we have not handled the exception, any exception raised will be caught by default java exception handler, printing the error message and terminating the program.

14.7 Handling checked exception

- Checked Exceptions forces programmers to deal with the exception that may be thrown
- When a checked exception occurs in a method, the method must either catch the exception and take the appropriate action, or pass the exception on to its caller
- IOException is checked exception
- "checked" means they will be checked at compile time itself
- You should compulsorily handle the checked exceptions in your code, otherwise your code will not be compiled

<u>throws</u>

- The throws keyword in java programming language is applicable to a method to indicate that the method raises particular type of exception while being processed.
- The throws keyword in java programming language takes arguments as a list of the objects of type java.lang.Throwables class.
- A **throws** clause can be used in the method prototype

  Method() throws <ExceptionType**1**>,…, <ExceptionType**n**>

  {

  }

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception
- A throws clause lists the types of exceptions that a method might throw.
- This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses.

Find output of the program

```
class ExceptionExample10{
  public void fun(){
    throw new IllegalAccessException();
  }
  public static void main(String[] args){
    ExceptionExample10 Obj=new ExceptionExample10();

    Obj.fun();
    System.out.println("hello");

    }
}
```

- In previous example, a compile time error is generated
- Compiler produces error as kind of exception is **checked exception** and it must be either caught or declared to be thrown.
- Use of throws to overcome this problem.
- Alternatively, try catch block can also get rid of problem.

Improved Example

```
class ExceptionExample11{
  public void fun() throws IllegalAccessException{
    throw new IllegalAccessException();
  }
  public static void main(String[] args){
    ExceptionExample11 Obj=new ExceptionExample11();

    Obj.fun();
    System.out.println("hello");
    }
}
```

- Oops the problem is yet not fully solved.
- By using throws, we just told that the method might generate such exception, but the method is invoked from main function….so either you should use throws for main method or use try block in main method.
- Following is the correct way of handling checked exception:

```
class ExceptionExample12{
  public void fun() throws IllegalAccessException{
    throw new IllegalAccessException();
  }
  public static void main(String[] args){
    ExceptionExample12 Obj=new ExceptionExample12();
    try{    Obj.fun();      }
    catch(Exception e){System.out.println("Error caught");}
    System.out.println("hello");
    }
}
```

Check Yourself:

Objective Type Question:

1. If we write return in the try block then the finally block will execute ?

A. Yes.

B. No.

2. Consider the code:

```
 try
    {
//.....
       }
   catch(Exception ex)
   {
//.....
         }
catch(NumberFormatException e)
{
//.....
}
```

What will occur if we execute this code?

A. will not compile.

B. exception.

C. code will execute.

D. none of the above.

3. If some code within a method throws a checked exception, then.. choose the appropriate option:

a. try-catch block is compulsory.

b. method must specify the exception using *throws* keyword.

c. try-catch block is optional.

A. a and b.

B. either a or b.

C. a or c.

D. c

4. How we create user defined exceptions?

A. by extending Object class.

B. by extending the class Exception or one of its subtypes.

C. by extending the class Error.

D. we cannot create user defined exceptions.

5. What will happen if we do not use try catch block in case of unchecked exception?

A. exception.

B. it depends upon the execution of the code.

C. code will not compile.

D. none of the above.

6. Consider the given code:

try { int x = Integer.parseInt("two"); }

Which could be used to create an appropriate catch block?

A. ClassCastException.

B. IllegalStateException.

C. IllegalArgumentException.

D. ArrayIndexOutOfBoundsException.

7. What will you do if you don't know what exception and how may exceptions will occur in your code?

A. will write a number of catch blocks

B. will catch exception as catch(Exception ex)

C. you will have to

8. What will happen if we write try without catch block but with finally block?

A. it will compile.

B. will not compile.

C. will compile but will not be executed.

D. none of the above.

9. If we write System.exit() in the try block then the finally block will execute ?

A. Yes.

B. No.

C. Maybe

D. None.

10. What is the output of this program?

```
class exception_handling {
  public static void main(String args[]) {
        try {
              throw new NullPointerException ("Hello");
              System.out.print("A");
        }
        catch(ArithmeticException e) {
              System.out.print("B");
        }
  }
```

Subjective Type Question:

1. Write a java code to take input using Scanner class methods and give string input to the method nextInt(). Now handle the exception with proper message.

2. Write a program to accept two integers from the user and pass them to the method addNos() which add two numbers. If any of the two numbers is zero then generate user defined exception.

3. Write a program to create your exception subclass that throws exception if the sum of two integers is greater than 99.

4. Write a java code to accept two numbers from the user and calculate their division in the user defined function called divNos(). If the denominator is entered as zero then handle the exception in main method.

5. Write a java code to generate ArithmeticException, NumberFormatException, and InputMismatchException in one try block and handle it by using only one catch block.

6. Write a java code to accept two integers from the user and calculate their sum and division. If the denominator is 0 then handle the exception in such a way that the sum should always be printed.

7. Write a program that shows that the order of catch blocks is important. If you try to catch a superclass exception type before a subclass type, the compiler should generate

errors.

8. Define an object reference and initialize it to null. Try to call a method through this reference. Now wrap the code in a try-catch clause to catch the exception.

9. Write a java code to sort the given set of n integers in descending order. Include a try block to locate the array index out of bounds exception and catch it.

10. Write a java code that repeatedly prompts the user to enter a number at the command line. It stops when a non numeric value is read in.

Chapter 15: Threading

15.1 What are Threads?

1) A thread is an independent path of execution within a program.
2) Many threads can run concurrently within a program
3) Multithreading refers to two or more tasks executing concurrently within a single program.
4) Every thread in Java is created and controlled by the **java.lang.Thread class**
5) There are two ways to create thread in java;
    a. Implement the Runnable interface (java.lang.Runnable)
    b. By Extending the Thread class (java.lang.Thread)

15.2 Creating Thread via Runnable Interface

- One way to create a thread in java is to implement the Runnable Interface and then instantiate an object of the class

- Runnable Interface has only one abstract method run().

public interface Runnable {

```
    void run();
}
```

- We need to override the run() method into our class which is the only method that needs to be implemented
- An object of Thread class is created by passing a Runnable object as argument to the Thread constructor. The Thread object now has a Runnable object that implements the run() method.
- The start() method is invoked on the Thread object created in the previous step. The start() method returns immediately after a thread has been spawned
- The thread ends when the run() method ends, either by normal completion or by throwing an uncaught exception

Example

```
class A implements Runnable
{
 public void run()
 {
  int i;
    try{
  for(i=1;i<=10;i++)
  {
  System.out.println("Hello "+i+Thread.currentThread());
     Thread.currentThread().sleep(10);
  }
  }catch(Exception e) {}

 }
}
```

```
class B implements Runnable
{
 public void run()
 {
  int j;
  try{
  for(j=11;j<=20;j++)
  {
   System.out.println("Bye "+j+Thread.currentThread());
   Thread.currentThread().sleep(1000);
  }
  }catch(Exception e){}
 }
}

public class ThreadExample1
{
 public static void main(String[] args)
 {
    A t1=new A();
    B t2=new B();
    t1.start();
    t2.start();
 }
}
```

- **currentThread()** is a static method of Thread class that returns reference of current thread object.
- **getName ()** is a member method of Thread class which returns a string telling the name of the thread.
- **sleep(long milli)** is a static method of Thread class that takes milliseconds as an argument. It cease the execution of thread for specified time
- This approach of creating a thread by implementing the Runnable Interface must be used whenever the class being used to instantiate the thread object is required to extend some other class.

15.3 Creating Thread via Thread Class

- A class extending the Thread class overrides the run() method from the Thread class to define the code executed by the thread
- This subclass may call a Thread constructor explicitly in its constructors to initialize the thread, using the super() call.
- The start() method inherited from the Thread class is invoked on the object of the class to make the thread eligible for running

Example

```
class A extends Thread
{
 public void run()
 {
  int i;
    try{
  for(i=1;i<=10;i++)
  {
  System.out.println("Hello "+i+Thread.currentThread());
     Thread.currentThread().sleep(10);
  }
  }catch(Exception e) {}

 }
}
class B extends Thread
{
 public void run()
 {
  int j;
  try{
  for(j=11;j<=20;j++)
  {
   System.out.println("Bye "+j+Thread.currentThread());
   Thread.currentThread().sleep(1000);
  }
  }catch(Exception e){}
 }
}
```

```
public class ThreadExample1
{
 public static void main(String[] args)
 {
    A t1=new A();
    B t2=new B();
    t1.start();
    t2.start();
    System.out.println("main"+Thread.currentThread());
 }
}
```

15.4 Thread States

- A Java thread is always in one of several states which could be running, sleeping, dead, etc.
- States:
    - New thread
    - Runnable
    - Not Runnable
    - Dead

New Thread

- A thread is in this state when the instantiation of a **Thread** object creates a new thread but does not start it running.
- A thread starts life in the Ready-to-run state.
- You can call only the **start()** or **stop()** methods when the thread is in this state.
- Calling any method besides **start()** or **stop()** causes an IllegalThreadStateException. (A descendant class of RuntimeException)

Runnable

- When the **start()** method is invoked on a New Thread() it gets to the runnable state or running state by calling the run() method.
- A Runnable thread may actually be running, or may be awaiting its turn to run.

Not Runnable

- A thread becomes Not Runnable when one of the following four events occurs:
- When **sleep()** method is invoked and it sleeps for a specified amount of time
- When **suspend()** method is invoked
- When the **wait()** method is invoked and the thread waits for notification of a free resource or waits for the completion of another thread or waits to acquire a lock of an object
- The thread is blocking on I/O and waits for its completion

Switching from not runnable to runnable state

- If a thread has been put to sleep, then the specified number of milliseconds must elapse (or it must be interrupted).
- If a thread has been suspended, then its resume() method must be invoked
- If a thread is waiting on a condition variable, whatever object owns the variable must relinquish it by calling either notify() or notifyAll().
- If a thread is blocked on I/O, then the I/O must complete.

Dead State

- A thread enters this state when the **run()** method has finished executing or when the **stop()** method is invoked. Once in this state, the thread cannot ever run again.

15.5 Thread Priority

- In Java we can specify the priority of each thread relative to other threads
- Those threads having higher priority get greater access to available resources then lower priority threads.
- A Java thread inherits its priority from the thread that created it
- You can modify a thread's priority at any time after its creation using the setPriority() method and retrieve the thread priority value using getPriority() method.
- The following static final integer constants are defined in the Thread class:

- MIN_PRIORITY (0) Lowest Priority
- NORM_PRIORITY (5) Default Priority
- MAX_PRIORITY (10) Highest Priorit

15.6 Synchronization

- 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 result due to concurrency issue.
- 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.
- Java programming language provides a very handy way of creating threads and synchronizing their task by using **synchronized** blocks.
- You keep shared resources within this block.

```
synchronized (objectidentifier) {

// Access shared variables and other shared resources

}
```

- You can also synchronized instance method

```
public synchronized void methodName(arguments)
{
 //code
}
```

- The use of the synchronized keyword in the method declaration. This tells Java that the method is synchronized.
- A synchronized instance method in Java is synchronized on the instance (object) owning the method. Thus, each instance has its synchronized methods synchronized on a different object: the owning instance. Only one thread can execute inside a synchronized instance method. If more than one instance exist, then one thread at a time can execute inside a synchronized instance method per instance. One thread per instance.

15.7 The Producer – Consumer Problem

The producer-consumer problem is one of the most frequently encountered problems when we attempt multi threaded programming. The main requirements are:

1. There are two different threads , called Producer and Consumer respectively, running and sleeping for random period of time.
2. The Producer thread produces an integer value and stores it in a variable.
3. The Consumer thread has to read this value and print it on the screen.
4. **The Producer should not produce new value until previous value is consumed by the consumer and the Consumer thread should not consume same value twice**

   Following is the first attempt to this problem

```
class SharedData
{
private int x;
synchronized void put(int i)
{
x=i;
System.out.println("Produced :"+x);
}
```

```
synchronized int get( )
{
System.out.println("Consumed :"+x);
return x;
}
}
class Producer implements Runnable
{
Thread th;
SharedData S;
```

```
public Producer(SharedData obj)
{
S=obj;
th=new Thread(this);
th.start();
}
public void run()
{
try
{
  for(int i=1;i<=10;i++)
  {
  Thread.sleep((int)(Math.random()*3000));
  S.put(i);
  }
}
catch(InterruptedException ex)
{
System.out.println("Producer Interrupted");
}
}
}
```

```
class Consumer implements Runnable
{
Thread th;
SharedData S;
public Consumer(SharedData obj)
{
S=obj;
th=new Thread(this);
th.start();
}
public void run( )
{
try
{
int sum=0,v;
do
{
Thread.sleep((int)(Math.random()*3000));
   v=S.get( );
  sum=sum+v;
}while(v!=10);
System.out.println("Sum of Consumed Data is "+sum);
}
catch(InterruptedException ex)
{
System.out.println("Producer Interrupted");
}
```

```
        }
        }

        class ProdCon
        {
        public static void main(String [ ] args)
        {
        SharedData obj=new SharedData( );
        Producer P=new Producer(obj);
        Consumer C=new Consumer(obj);
        try
        {
        P.th.join();
        C.th.join();
        }
        catch(InterruptedException ex)
        {
        System.out.println("Main Interrupted");
        }
        }
}
```

Output:

Consumed :0

Produced :1

Consumed :1

Consumed :1

Consumed :1

```
Consumed :1

Consumed :1

Produced :2

Consumed :2

Consumed :2

Produced :3

Consumed :3

Produced :4

Consumed :4

Produced :5

Consumed :5

Produced :6

Consumed :6

Consumed :6

Produced :7

Produced :8

Produced :9

Consumed :9

Consumed :9

Produced :10

Consumed :10

Sum of Consumed Data is 61
```

Now it can be observed that output is not correct as Consumer has consumed some data multiple times and producer has produced new data without previous data getting consumed . This has happened because both Producer and Consumer are not waiting for each other to complete it's task. To solve this problem we need to use the methods **wait( )** and **notify( )** belonging to **Object** class.

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.

Modified SharedData class:

```
class SharedData
{
private int x;
private boolean done=true;
synchronized void put(int i)
{
if(done==false)
{
 try
 {
      wait();
 }
 catch(InterruptedException ex)
 {
      System.out.println("Producer Interrupted!");
 }
}
x=i;
System.out.println("Produced :"+x);
done=false;
notify();
```

```
}
synchronized int get( )
{
if(done==true)
{
 try
 {
      wait();
 }
 catch(InterruptedException ex)
 {
      System.out.println("Consumer Interrupted!");
 }
}
System.out.println("Consumed :"+x);
done=true;
notify();
return x;
}
}
```

Output:

Produced :1

Consumed :1

Produced :2

Consumed :2

Produced :3

Consumed :3

Produced :4

Consumed :4

Produced :5

Consumed :5

Produced :6

Consumed :6

Produced :7

Consumed:7

Produced :8

Consumed:8

Produced :9

Consumed :9

Produced :10

Consumed :10

Sum of Consumed Data is 55

Check Yourself:

Objective Type Question:

1. How many methods are available in runnable interface?

A. 1

B. 2

C. 3

D. 0

2. What is the base class of Thread?

A. Runnable

B. Object.

C. String.

D. Threadsafe.

3. How many priority constants are there ?

A. 2

B. 4

C. 1

D. 3

4. Multitasking and Multithreading are the features of :

A. Programming Language and OS.

B. OS and Programming Language.

C. Programming Language and Programming Language.

D. OS and OS.

5. Choose the correct statement about daemon thread:

a. jvm quits when the execution of non daemon threads is completed.

b. jvm will wait for the daemon threads to execute.

c. the main thread which runs the main() is by default non daemon.

A. a and b.

B. a and c.

C. all of the above.

D. none of the above.

---

6. Benefits of Multithreading can be:

a. great responsiveness.

b. resource sharing.

c. utilization of multiprocessors.

A. a and c.

B. a and b.

C. a and c but not b.

D. all of the above.

7. What is Multithreading?

A. technique that allows a program or a process to execute many tasks concurrently.

B. It allows a program to be more responsible to the user

C. Both.

D. Only A.

8. Choose the right option.

A. The notifyAll() method must be called from a synchronized context.

B. To call wait(), an object must own the lock on the thread.

C. The notify() method is defined in class java.lang.Thread.

D. The notify() method causes a thread to immediately release its lock.

9. A thread becomes not runnable when..

a. sleep () is invoked.

b. thread calls the wait() to wait for the specific condition to be satisfied.

c. thread is blocking on I/O.

A. both a and c.

B. both b and c.

C. none of the above.

D. all of the above.

10. Which of the following methods have overloaded versions?

A. sleep().

B. sleep() and yield().

C. join().

D. sleep() and join().

Subjective Type Question:

1)WAP to accept 2 int from user now make 2 thread which will do addition and subtraction of these no's.

2)WAP to make 2 thread one thread read the file and another thread write the content read by first thread into another file.

3)WAP to Accept String from user now make two threads first thread print the string as it is second thread will print it in reverse order.

4)make 2 threads one thread generate the no's radomly and as soon as first thread genereate the two no's second thread will add them and disply their sum on console.

5)WAP to draw a small circle on a frame now make a thread which changes the color of circle after every one second.

6)WAP to make a frame which should change it's color after every 10 second.

7)WAP to make a frame in which (bullet) a points randomly changes its position using thread.

8)WAP in which make two threads first thread gets Sting input from user and second thread displays it but until first thread has not got input from user second thread displays "wait" on console as soon as input is acceptes second thread will display the input.

---

9)Wap in which make 2 threads and set priorities of them using constants available in Thread class now execute it and show the msg on console that what thread is currently executes.

10)make a thread which according to u gets suspended and resumed as soon as it starts execution it prints no's stating from 1 until it gets suspended and whenever it get resume again print the no's from where it left.

Chapter 16: File Handling

16.1 Java IO

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java.
All these streams represent an input source and an output destination
Java provides strong but flexible support for I/O related to Files
File handling is an integral part of nearly all programming projects
Files provide the means by which a program stores data

16.2 Streams

- A stream is an ordered sequence of bytes of arbitrary length.
- Bytes flow over an **output stream** from an application to a destination, and flow over an **input stream** from a source to an application.
- Java recognizes various stream destinations; for example, byte arrays, files, screens, and sockets (network end points).
- Java also recognizes various stream sources; for example, byte arrays, files, keyboard, and sockets.
- Modern versions of Java define two types of streams: **byte** and **character**

- Byte streams provide a convenient means for handling input and output of bytes.
- Character streams are designed for handling the input and output of characters

Byte Streams

- Byte streams are defined by two class hierarchies: one for input and one for output.
- At the top of these are two abstract classes: **InputStream** and **OutputStream**.
- InputStream defines the characteristics common to byte input streams, and OutputStream describes the behavior of byte output streams
- Some of the byte stream classes are:
    - BufferedInputStream
    - BufferedOutputStream
    - ByteArrayInputStream
    - ByteArrayOutputStream
    - DataInputStream
    - DataOutputStream
    - FileInputStream
    - FileOutputStream
    - PrintStream
    - RandomAccessFile

Character Stream

- Java Byte streams are used to perform input and output of 8-bit bytes, where as Java Character streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are , FileReader and FileWriter.

- Some of the character stream classes are:

    - BufferedReader
    - BufferedWriter
    - CharArrayReader
    - CharArrayWriter
    - FileReader
    - FileWriter
    - InputStreamReader
    - OutputStreamWriter
    - PrintWriter

- StringReader
- StringWriter

16.3 File Class

- File class is a representation of file and directory pathnames
- File class does not provide any read-write method.
- File class provides methods to access various characteristics of a file, like, file protection information, file size, existence of file etc

Example

```
import java.io.File;
class FileExample1{
  public static void main(String[] args){
    File f1=new File("f: /name.txt");
    System.out.println("Can file Read "+f1.canRead());
    System.out.println("Is file exist "+f1.exists());
    System.out.println("File name "+f1.getName());
    System.out.println("Length of file "+f1.length());
  }
}
```

- canRead() method returns Boolean value.
- exists() method returns a Boolean value, telling whether file exists or not.
- getName() method returns the name of a file. Return type is string.
- length() method returns an int value, telling the size of the file in bytes.

16.4 Reading content from a file

```
/* Reading data from file */
/* FileInputStream (Byte Stream Class) */
import java.io.*;
class FileExample3{
  public static void main(String []args) throws IOException{
    int i;
    FileInputStream fin;
    fin=new FileInputStream("name1.txt");
```

```
  do{
     i=fin.read();
     if(i!=-1)
       System.out.print((char)i);
  }while(i!=-1);
  fin.close();
  }
}
```

16.5 Writing content to a file

```
        /* Writing data to a file */
        /* FileOutputStream (Byte Stream Class) */
        import java.io.*;
        class FileExample4{
          public static void main(String []args) throws IOException{
            int i;
            FileOutputStream fout;
            fout=new FileOutputStream("name.txt",true); /* second argument is true,
        which means appending the content in the file. False means erasing the old
        content and writing as a fresh. */

            String s="Welcome Students";

            char ch[]=s.toCharArray();
            for(i=0;i<s.length();i++)
                  fout.write(ch[i]);
            fout.close();
          }
}
```

16.6 Reading/Writing an object from/to file

```
/* ObjectInputStream and ObjectOutputStream */


        import java.io.*;
        class Book implements Serializable
        {
                private String title;
            private double price;

            public void setTitle(String title)
            { this.title=title; }
            public String getTitle()
            { return(title); }
                public void setPrice(double price)
            { this.price=price;}
                public double getPrice()
            { return(price); }
        }
        public class FileExample
        {
        public static void reading() throws IOException, ClassNotFoundException{
           Book b2;
           ObjectInputStream ois=new ObjectInputStream(new
        FileInputStream("name.txt."));
           b2=(Book)ois.readObject();
           System.out.println("Title: "+b2.getTitle());
           System.out.println("Price: "+b2.getPrice());
         }
         public static void writing() throws IOException
         {
           Book b1=new Book();
           ObjectOutputStream oos=new ObjectOutputStream(new
        FileOutputStream("name.txt"));
           b1.setTitle("Core JAVA");
           b1.setPrice(350.50);
           oos.writeObject(b1);
```

```
        oos.close();
      }
    }
```

Check Yourself:

Objective Type Question:

1. What method is used to read a byte from System.in?

A. read()

B. readByte()

C. byteRead()

D. none of the above.

2. What method can be used to write to System.out?

a. print()

b. println()

c. write()

A. a and b.

B. a and c

C. b and c.

D. all of the above.

3. What does read() return when the end of file is encountered?

A. 1

B. 0

C. -1

D. eof.

4. readOject() and writeObject() are the methods of the.....classes respectively

A. ObjectInputStream and ObjectInputStream

B. ObjectInputStream and ObjectOutputStream

C. ObjectOutputStream and ObjectOutputStream

D. ObjectOutputStream and ObjectInputStream.

5. I/O operations of byte stream are handled by

A. abstract class InputStream.

B. abstract class OutputStream.

C. interface InputOutputStream.

D. A and B.

6. Input streams are used to read data from

a. a keyboard.

b. a disk file.

c. memory buffer

A. both a & b.

B. b & c.

C. only c

D. a, b & c.

7. What is Serialization?

A. It is the process of writing the state of an object to a byte stream.

B. It is the process of restoring these objects.

C. It is the process of reading an integer.

D. none of the above.

8. Select the correct option.

a. The InputStream and OutputStream classes are byte-oriented.

b.The ObjectInputStream and ObjectOutputStream do not support serialized object input and output.

c. The Reader and Writer classes are character-oriented.

d. The Reader and Writer classes are the preferred solution to serialized object output.

A. a and c

B. a and b

C. all of the above.

D. none of the above.

9. Choose the correct option regarding RandomAccessFile:

A. it extends the class OutputStream.

B. it access the contents of a file sequentially.

C. it implements the interface DataInput and DataOutput.

D. all of the above.

10. Choose the wrong option:

A. A stream is an abstraction that either produces or consumes information.

B. Java defines both byte and character streams.

C. System.err and System.out are not the built-in streams in java.

D. none of the above.

Subjective Type Questions:

1) WAP to Read a fileand display the content on console.

2) WAP to accept string from user and write into file.

3) WAP to copy one text file into another blank text file.

4) WAP to append two files and write all content into another file.

5) WAP to delete a specific word from file and store modified content into same file .
ex. "JAVA is pure object oriented language .its punch line is Wora".
remove the word "is" so sentence become "java pure object oriented language. its punch line wora"

6) WAP to add special charater (#) before every new line in a file.

7) WAP to swap the content of two files.

8) WAP to take a no from user and write even no between zero to that no in one file and odd no in another file.

9) Using printwriter print any msg on console.

10) WAP to accept an integer from the user and Write all the prime no between 2 to this no in a file.