Java file manipulations

Categories of Java errors

- We learn that there are three categories of Java errors :
  - **Syntax error**
  - **Runtime error**
  - **Logic error**.

- A Syntax error (compiler error) arises because a rule of the language has not been followed; they are detected by the compiler.

- Runtime errors occur while the program is running, if the environment detects an operation that is impossible to carry out.

- A logic error occurs when the program does not perform the way it was intended to.

Exception handling (טיפול בשגיאות)

- An **exception** (**חריגה**) is one of the abnormal conditions that can occur during the execution of a program.

  An **exception** is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

- An **exception** can occur for many reasons; some of the general reasons are the following:

  - A user has entered invalid data .
  - A file needs to be opened but cannot be found.
  - A network connection failed in the middle of communication.

Exception handling (טיפול בשגיאות)

In Java programming, were interested in two kinds of exceptions that might occur with a program :

1. **compile** (**syntax**) **errors**
2. **run – time errors**

1. A **compile error** usually occurs because the programmer made a fundamental mistake in the program, such as failing to include a semicolon at the end of a statement.

```
public class Point
{
 private double  x
 private double  y;
```

⇦ **Compile error**

2. A **run – time error** occurs when the program runs and is caused by a number of reasons, such as dividing by zero.

```
    public static void main(String[] args) {
        int a=10;
        int b=0;
        System.out.println(a/b);
    }
```

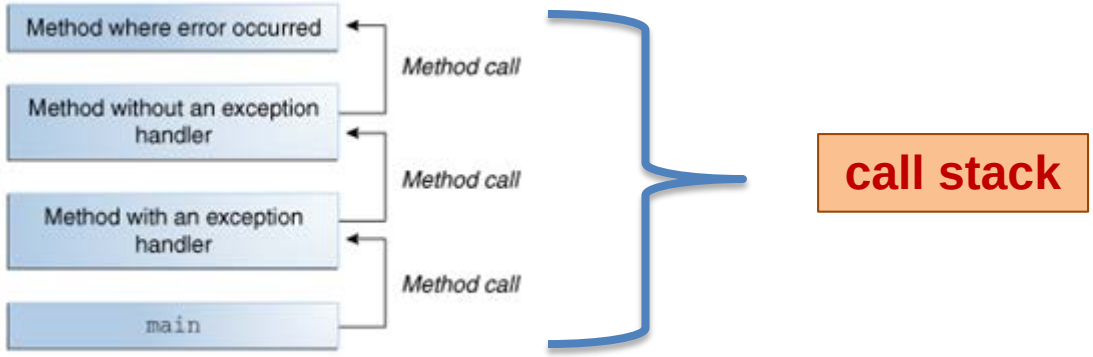
Problems | Javadoc | Declaration | Console

```
<terminated> Rec [Java Application] C:\Program Files (x86)\Java\jre6\bin\javaw.exe (21/11/2012 16:27:36)
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Rec.main(Rec.java:48)
```

⇦ **Run-time error**

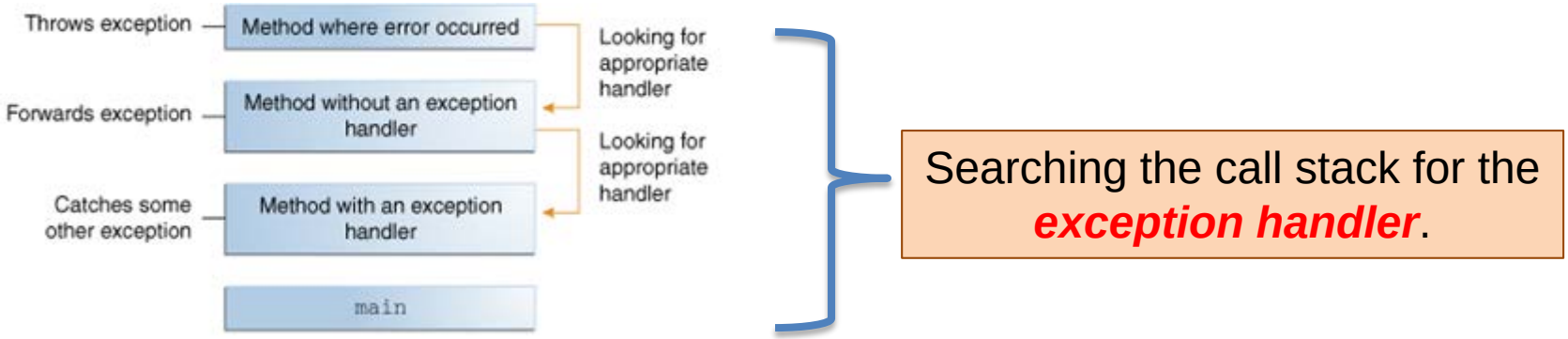
Exception handling (טיפול בשגיאות)

- When an error occurs within a method, the method *creates an object* and hands it off to the runtime system.
- The object, called an ***exception object***, contains information about the error, including its type and the state of the program when the error occurred.
- After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "something" is the ordered list of methods that had been called to get to the method where the error occurred.
  The list of methods is known as the **call stack**.



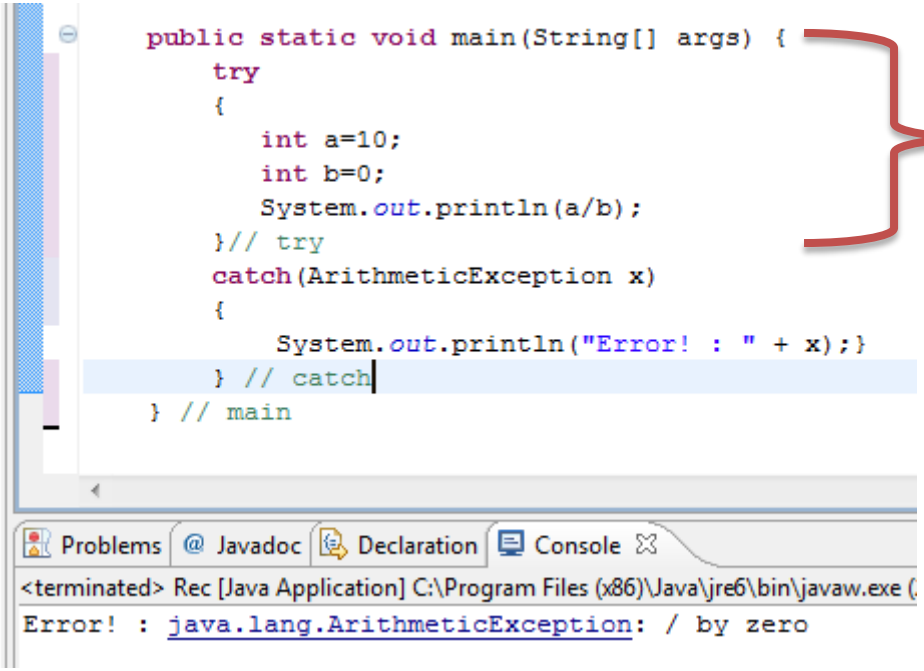
Exception handling (טיפול בשגיאות)

- The runtime system searches the ***call stack*** for a method that contains a block of code that can handle the exception.
  This block of code is called an ***exception handler***.
- The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called.
- When an appropriate handler is found, the runtime system passes the exception to the handler. The exception handler chosen is said to ***catch the exception***.



Exception handling (טיפול בשגיאות)

- Programmers build into their programs *exception handlers* designed to react to ***run-time errors only***.
- An *exception handlers* is a portion of the program that contains statements that execute automatically whenever a specific run-time error occurs while the program runs. This is referred to as *catching* (*תפיסה*) an exception.
- Statements that you want monitored by Java must appear within a ***try block***.



try block

catch block

Statements that are executed when an exception is thrown are placed within a ***catch block***.

Try and catch blocks

- The first step in constructing an exception handler is to enclose the code that might throw an exception within a try block.
  In general, a **try block** looks like the following:

```
try  {
       …code
     }
catch or finally blocks . . .
```

If an exception occurs within the **try block**, that exception is handled by an exception handler associated with it.
To associate an exception handler with a try block, **you must** put a **catch block** after it.

Note: No code can be between the end of the try block and the beginning of the first catch block !

Try and catch blocks

In general, a **catch block** looks like the following:

catch (ExceptionType name) { …code }

- Each catch block is an exception handler and handles the type of exception indicated by its argument.
  The argument type, **ExceptionType**, declares the type of exception that the handler can handle and must be the name of a **Java exception class**.
  (see next slide)

- The handler can refer to the exception with **name** parameter.

- The catch block contains code that is executed if and when the exception handler is invoked.

Java exception classes

- ArithmeticException
- FileNotFoundException
- **IOException**
- NumberFormatException
- ArrayIndexOutBoundsException
- ArrayStoreException
- Illegal ArgumentException
- ClassCastException

Basic exception handling

- A catch block responds to **one kind of exception**, which is specified within the catch block's parentheses.
- Every **try block must** have at least one **catch block** or a **finally block**.
- The **catch block must** appear immediately following its corresponding try block. Failure to pair them causes a compiler error.
- In the real programming a series of statements might generate more than one kind of run-time error.
- **Multiple catch block** are used to respond to multiple exceptions.
- **Multiple catch block must** immediately follow the try block where the exception might be thrown.
- Also, each of those catch blocks **must** follow one another.

Multiple catch block - example

```
public static void main(String[] args) {
    try
    {
       int a[] = new int[3];
       a[0]=10;
       a[1]=0;
       a[2]=a[0] / a[3];
    }// try
    catch (ArithmeticException x1)
    {
        System.out.println("Error! : " + x1);
    } // Arithmetic catch
    catch (ArrayIndexOutOfBoundsException x2)
    {
        System.out.println("Error!!! : " + x2);
    } // ArrayIndex catch
} // main
```

First catch block

Second catch block

Problems | Javadoc | Declaration | Console

<terminated> Rec [Java Application] C:\Program Files (x86)\Java\jre6\bin\javaw.exe (21/11/2

```
Error!!! : java.lang.ArrayIndexOutOfBoundsException: 3
```

Exception, which is thrown if the program uses an index that is out of bounds of the array – index 3.

Finally block

The **finally block** is used to place statements that must execute regardless of whether an exception is not thrown.
That is, statements within the finally block execute **all the time !**

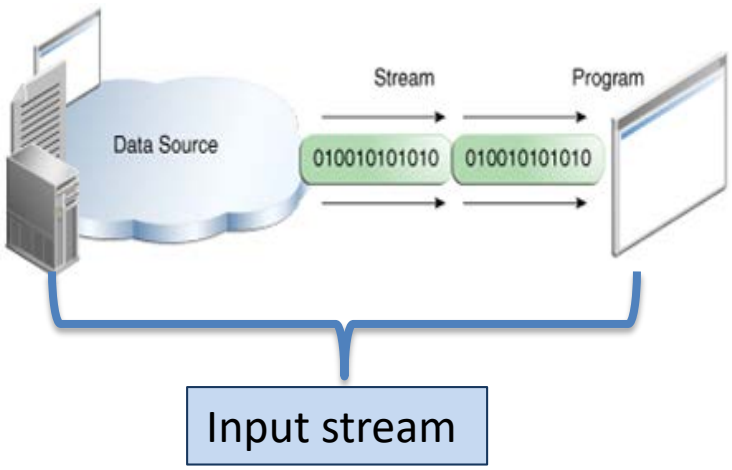
```
public static void main(String[] args) {
    try
    {
       int a[] = new int[3];
       a[0]=10;
       a[1]=0;
       a[2]=a[0] / a[3];
    }// try
    catch (ArithmeticException x1)
    {
        System.out.println("Error! : " + x1);
    } // Arithmetic catch
    catch (ArrayIndexOutOfBoundsException x2)
    {
        System.out.println("Error!!! : " + x2);
    } // ArrayIndex catch
    finally {
        System.out.println("The finally block executed." );
    } // finally block
} // main
```

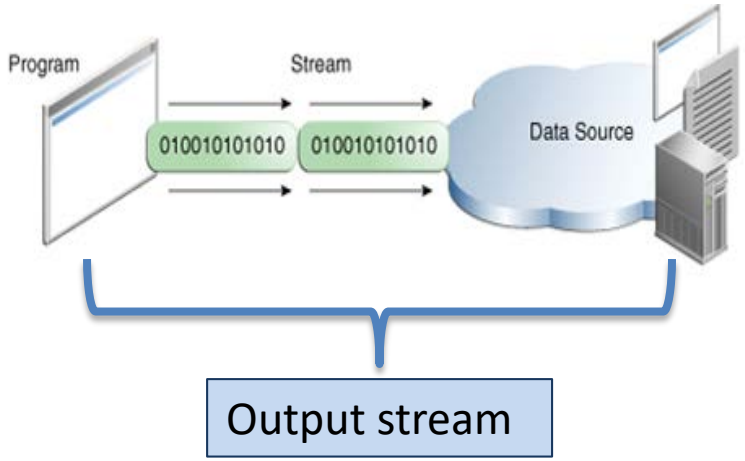
```
Problems  Javadoc  Declaration  Console
<terminated> Rec [Java Application] C:\Program Files (x86)\Java\jre6\bin\javaw.exe (21/11/2012 18:54:31)
Error!!! : java.lang.ArrayIndexOutOfBoundsException: 3
The finally block executed.
```

This example is the same as the previous example, except we included the **finally block**.

Input / Output streams

- **A stream** (**זרם**) is an ordered sequence of bytes. It can be used as a source of input or a destination for output.
- A stream represents a flow of data, or a channel of communication with a writer at one end and a reader at the other.
- In a Java program, we treat a stream as either an **input stream**, from which we **read** information, or as an **output stream**, to which we **write** information.
- A particular store of data, such as **a file**, can serve either as an input stream or as an output stream to a program, but is cannot be both at the same time. **Streams in Java are one-way streets**.
- Java provides many classes that let us define streams in different ways.





Standard Input / Output

- Three streams are often called the **standard I/O streams** :

Standard I/O Stream	Description
System.in	Standard input stream
System.out	Standard output stream
System.err	Standard error stream (Output for error messages)

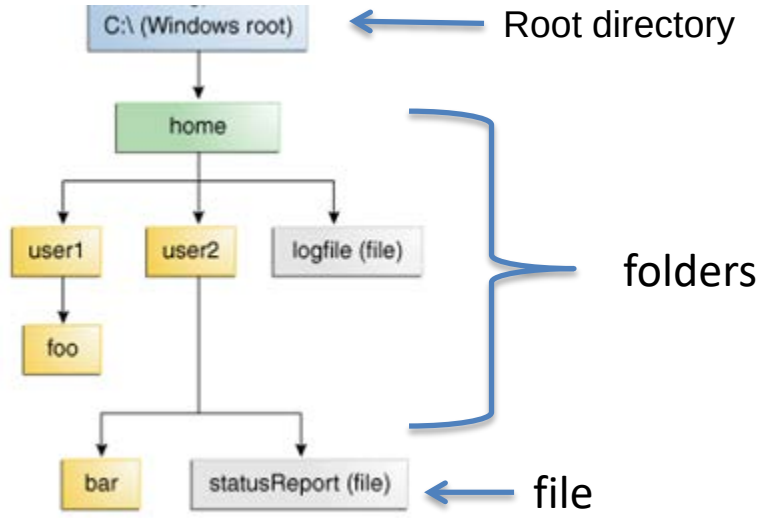
- The **System class** contains three object reference variables(**in**, **out** and **err**) that represent the three standard I/O streams.
- The standard I/O streams ,by default, represent particular I/O devices.
  - System.**in** represents keyboard input.
  - System.**out** and System. err represents monitor screen.

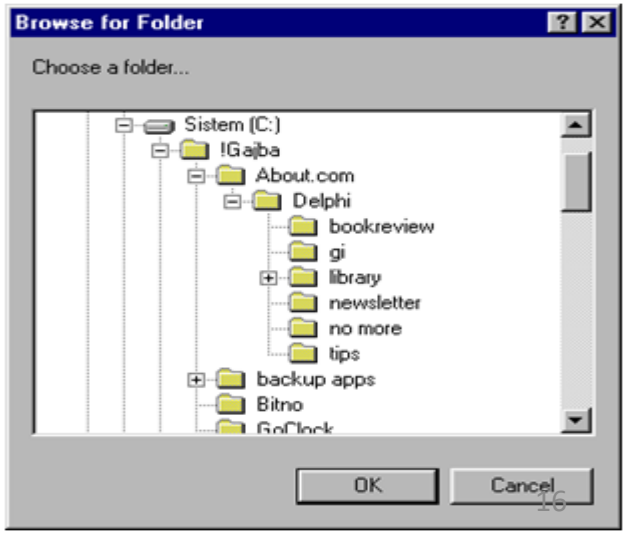
For example: System .out.println("Stream example");

- All three of these streams are created and opened by default.

Files and streams

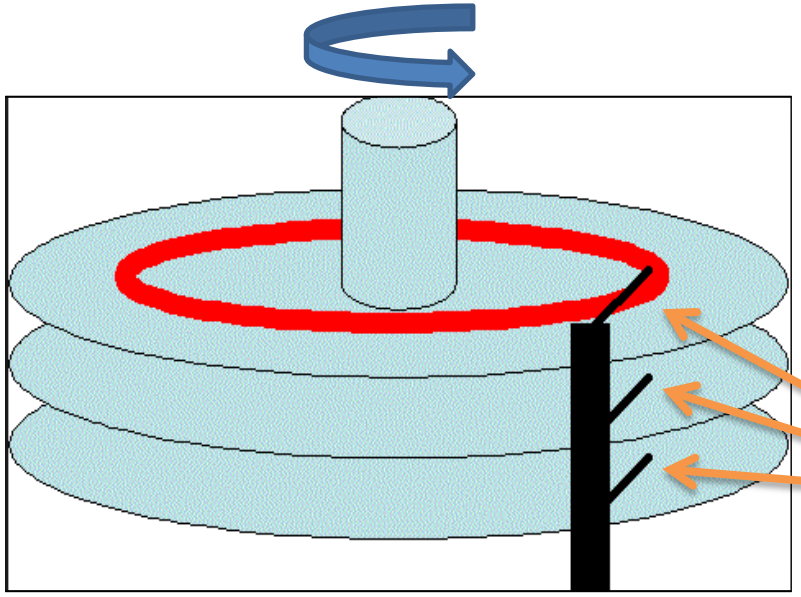
- A **file** (**קובץ**) is a logical grouping of related bytes stored in secondary storage. Java views each file as a ***sequential stream*** (***זרם סידרתי***) ***of bytes***.
- A **file system** is software used to organize and maintain files on second storage device.
- Each file has properties that describe the file(name, size, date when the file was last update).
- File are organized in a series of directories and subdirectories, called **folders**. The topmost directories is called **root directory**. Each root node maps to a volume, such as C:\ or D:\





Hard disk drive (HD)

- The **hard drive** (**HD**) is the computer's main storage media device that permanently stores all data on the computer.
- The hard drive was first introduced on September 13, 1956 and consists of one or more hard drive platters inside of air sealed casing.
- When the operating system needs to read or write information, it examines the hard drives **F**ile **A**llocation **T**able (**FAT**) to determine file location and available areas.
- Java provides a standard way of reading from and writing to files.
  The **java.io package** contains classes which can be used to read and write files.



Platters : one or more ceramic disk that is coated in a magnetic media.

read/write **heads**

Hard disk drive (HD)

The File class

- The **File class** offers a rich set of static methods for reading, writing, and manipulating files and directories.

  Using the File class we can:
- To create an ***instance*** of a **File class** we use one of two constructors :

1. File file1 = new File(String directory);

for example: c:\temp\lectures is a directory path (נתיב) that leads to the lectures subdirectory within the temp directory of the c drive.

```
File file1 = new File("c:\temp\lectures");
```

2. File file2 = new File(String directory , String fileName);

The first parameter is the directory path and the second parameter is the name of a file contained in the last subdirectory of the path.

for example: File file2 = new File("c:\temp\lectures","lec1.doc");

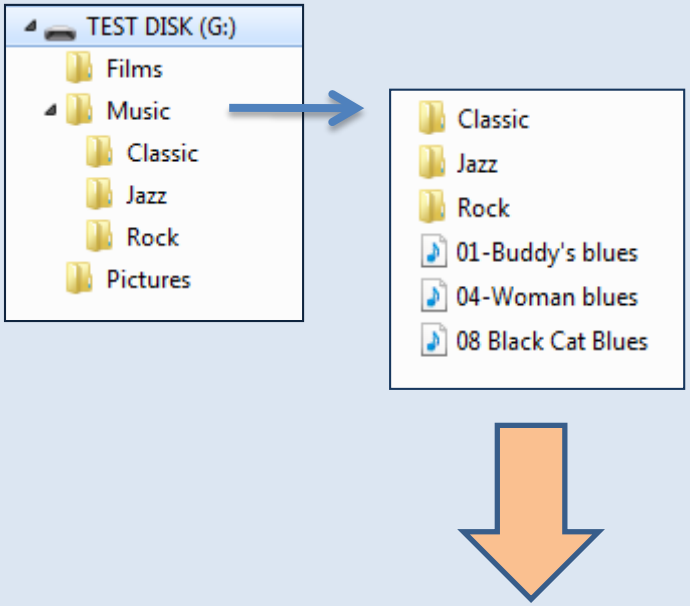
The File class - methods

Note: These methods do not create a directory or subdirectory, nor do they create a file. Instead, these methods **are a pointing** to either a directory path or a file.

Method	Description
isFile()	Returns true if the object is a file, otherwise a false is returned.
delete()	Deletes the file.
length()	Returns the length of the file in bytes.
exists()	Returns true if the directory path or file exists, otherwise a false is returned.
getParent()	Return the name of the parent directory
isDerectory()	Return true if it is a directory, otherwise a false returned.
list()	Return an array of strings containing the names of the files stored in the directory.

The File class - example

```
public static void main(String[ ] args)  {
      String dir = "G:\Music";
      File f1 = new File(dir);
      if(f1.isDirectory())
      {
            System.out.println( "Directory :" + dir);
            String arr[ ] = f1.list();
            for(int i = 0; i < arr.length; i++)
            {
                  File f2 = new File(dir + "\" + arr[i]);
                  if(f2.isDirectory())
                        System.out.println( "Directory: " + arr[i]);
                  else
                        System.out.println( "File: " + arr[i]);
            } // for
      } // if
      else
            System.out.println( "Not a directory" );
   } // main
```

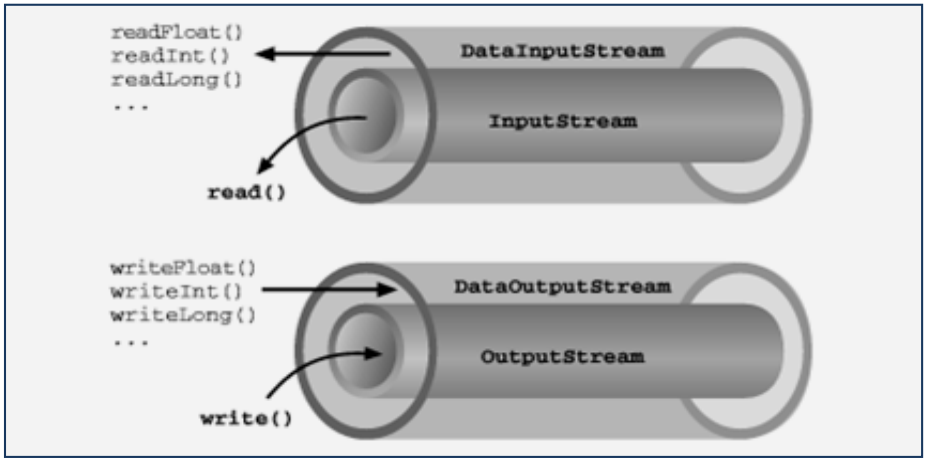



```
Directory :G:\Music
Directory: Classic
Directory: Jazz
Directory: Rock
File: 01-Buddy's blues.mp3
File: 04-Woman blues.mp3
File: 08 Black Cat Blues.wma
```

Files and streams

- In Java IO streams are flows of data you can either read from, or write to. Streams are typically connected to a data source, or data destination, like a file, network connection etc.

- A stream is just a continuous flow of data.



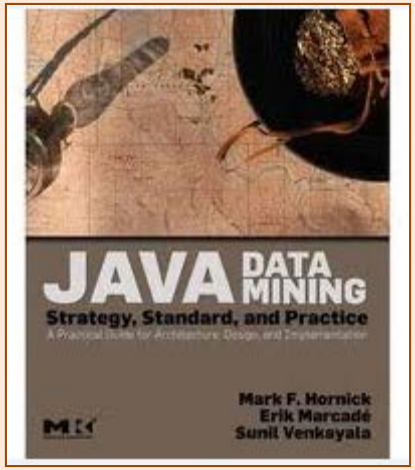
- In Java ***IO streams are typically byte based***. This means that you can either read bytes from, or write bytes to a stream.

- If you need to read / write characters (like UNICODE characters), you should use a **BufferReader** or **PrintWriter classes**.

Storing data in Java

Typically, Java program stores data on one of three ways :

- Individual pieces of data(like numbers, strings,arrays) that are not encapsulated in a class.
- Data that is encapsulated in a class(like Student, Point, Node).
- Data stored on a database.



Writing to a file

- Java has several stream classes that are built upon four execute basic file manipulations.
- In order to write data to a file , we need create a **File output stream**. When we send a stream of data to the **FileOutputStream**, data will be written to disk.
- A file output stream opens the file or creates a new file doesn't exist.
- Once the file output stream is open, we can write data to the file using a **PrintWriter**.
- We open a file output stream using the constructor of the **FileOutputStream** class and passing it the name of the file we want open. The constructor returns a reference to **PrintWriter**. We use the **PrintWriter** reference to call methods of the **PrintWriter** class to write to the file.

for example:

```
PrintWriter   outFile = new  PrinrWriter(new FileOutputStream("Student.dat"));
```

Writing to a file - example

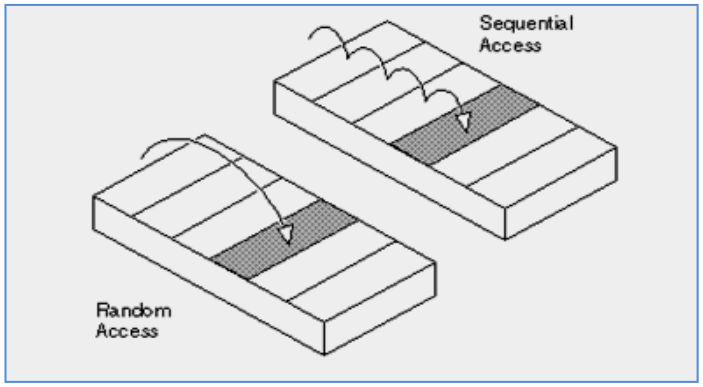
```
public static void main(String[ ] args)
 {
     try  {
            PrintWriter   outFile = new   PrintWriter( new  FileOutputStream( " e:/test.txt" ));
            System.out.println("Enter student first name-> ");
            String fName =reader.next();
            System.out.println("Enter student second name-> ");
            String sName =reader.next();
            System.out.println("Enter student address-> ");
            String addr =reader.next();
            outFile.print(fName + " ");
            outFile.print(sName + " ");
            outFile.println(addr);
            outFile.close();
     } //  try block
     catch(IOException  e)  {
            System.out.println( "Error I/O " + e);
     } // catch block
} // main
```

Note: that all statements involved in opening file output stream and writing to the file are contained within try block.

After finishing writing to the file, **we must** call the **close()** method to close the file.

Sequential access to the file

- In our course we use sequential access (גישה סידרתית) to the data, stored in a file.
- Java distinguishes between sequential-access data files and random-access (גישה אקראית) data files, allowing you to choose between the two types.
- Sequential-access files are faster if you always access data in the same order. Random-access files are faster if you need to read or write data in a random order.



- Devices can also be classified as sequential access or random access.

For example: a tape-drive is a sequential-access device . A disk drive , on the other hand, is a random-access device because the drive can access any point on the disk without passing through all intervening points.

Appending to a file

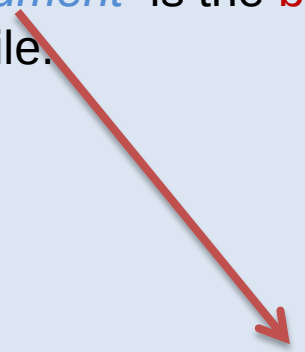
- In previously example, when we write data of the second student to the file " e:/test.txt" , the data will always written at the beginning of the file: the new data will overwrite the existing data in the file.

- Programmers usually want to add data to a file rather than replace existing data. To do this, new data must be written after the last byte in the existing file. We call this **appending data to a file**.

- FileOutputStream constructor uses to appending data to a file.
  Another version of the constructor uses two arguments.
  The *first argument* is again the filename, and the *second argument* is the boolean value **true**. This causes bytes to be written at the end of the file.

  The file is created if it doesn't exist.

  For example:



```
PrintWriter  outFile = new PrinrWriter(new  FileOutputStream("Student.dat", true));
```

Creating and Appending to a file - example

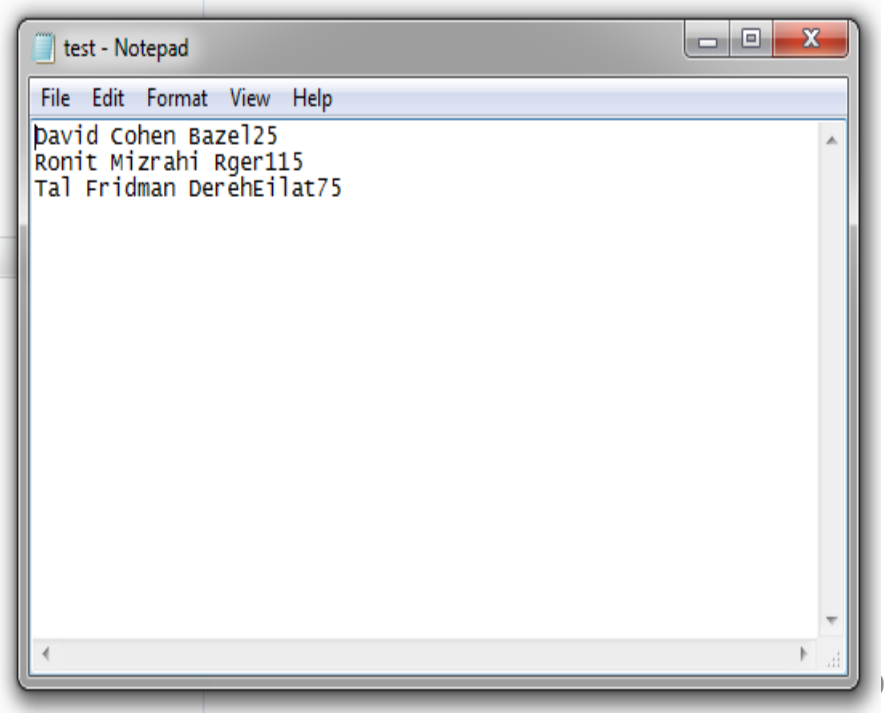
```
public static void main(String[ ] args)  {
    try  {
        PrintWriter  outFile = new  PrintWriter( new  FileOutputStream("e:/test.txt", true));
        for(int i = 0; i < 3; i++)  {
            System.out.println("Enter student first name-> ");
            String fName = reader.next();
            System.out.println("Enter student second name-> ");
            String sName = reader.next();
            System.out.println("Enter student address-> ");
            String addr = reader.next();
            outFile.print(fName + " ");
            outFile.print(sName + " ");
            outFile.println(addr);
        } // for
        outFile.close();
    } //  try
    catch(IOException  e)  {
        System.out.println("Error I/O " + e);
    } // catch
} // main
```

Appending to a file – execution

```
Enter student first name-> David
Enter student second name-> Cohen
Enter student address-> Bazel25
Enter student first name-> Ronit
Enter student second name-> Mizrahi
Enter student address-> Rger115
Enter student first name-> Tal
Enter student second name-> Fridman
Enter student address-> DerehEilat75
```

Input data

Output data



Reading from a file

- Java has several ways to read data from a file. We can read :
  - a byte of data at a time.
  - a specific number of bytes at the time.
  - a line of bytes at one time.

  A line consist of series of bytes that end with a byte that corresponds to a newline character.
- In order to read a file, we need to open the file. A common way is to create a file reader by using the constructor of the **FileReader class** and passing it the name of the file.
- It takes time to read bytes from a disk drive. In order to reduce this time, Java reads a chunk of bytes one time and store them in memory called a **buffer** (**חוצץ**). The program then reads bytes from the buffer instead of the disk drive.
- Java creates a buffer by using the **BufferedReader** constructor and passing it a reference to the **FileReader** used to open the file.

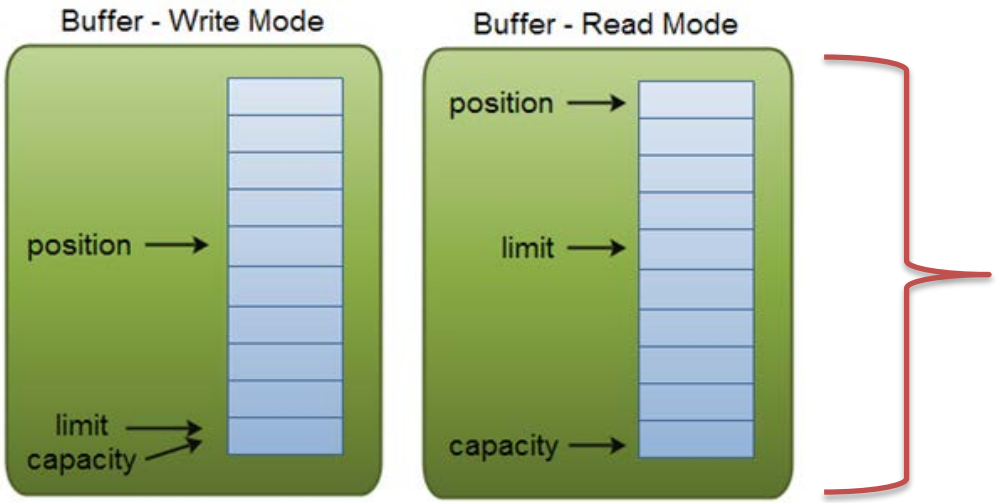
  For example:

  BufferedReader inFile = new BufferedReder(new FileReader("Student.dat"));

Buffers and I/O operations

A buffer is essentially a *block of memory* into which you can write data, which you can then later read again.
A Buffer has three properties you need to be familiar with, in order to understand how a Buffer works. These are:

- A buffer's *capacity* is the *number of elements* it contains. The capacity of a buffer is never negative and never changes.
- A buffer's *limit* is the *index of the first element* that should not be read or written. A buffer's limit is never negative and is never greater than the its capacity.
- A *buffer's position* is the *index of the next element* to be read or written. A buffer's position is never negative and is never greater than its limit.



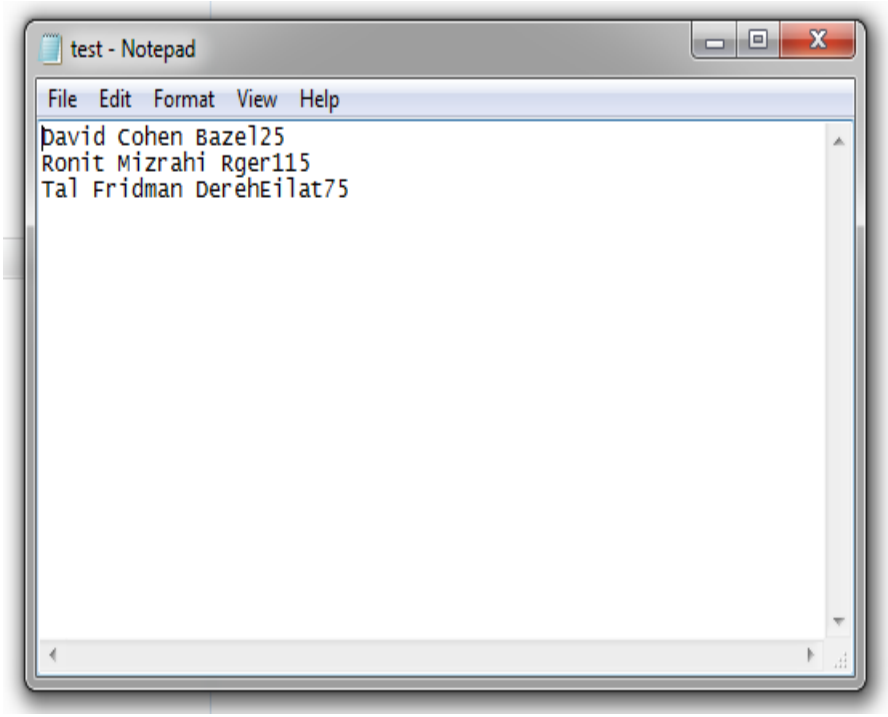
Buffer *capacity, position* and *limit* in write and read mode.

Reading from a file - example

```
public static void main(String[ ] args)
{
     String   line; // input string
     try {
          BufferedReader  inFile = new BufferedReader(new  FileReader("e:/test.txt"));
          while(( line= inFile.readLine()) != null)
                                   System.out.println(line);
          inFile.close();
     } // try  block
     catch(IOException  e)
     {
            System.out.println("Error I/O " + e);
     } // catch block
} // main
```

Reading from a file - execution

Input data



Output data

David Cohen Bazel25
Ronit Mizrahi Rger115
Tal Fridman DerehEilat75

Reading and Writing an object

- In the real programming, many data elements we want stored in a file will be data members of an object, such as student.

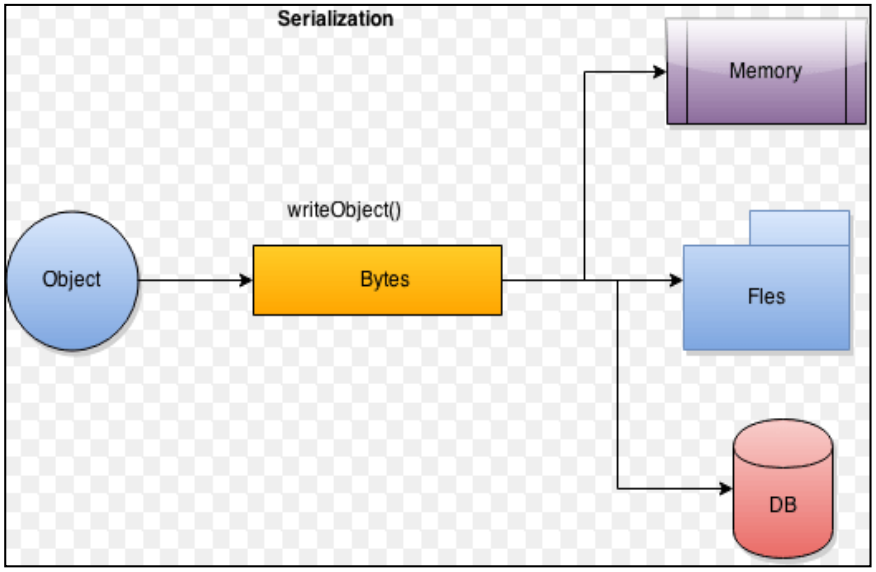
  Attributes are basically data, and behaviors are methods.

- Many attributes of a class are held in instance variables. When a programmer needs to retain an instance of a class, the programmer saves the instance to the file.

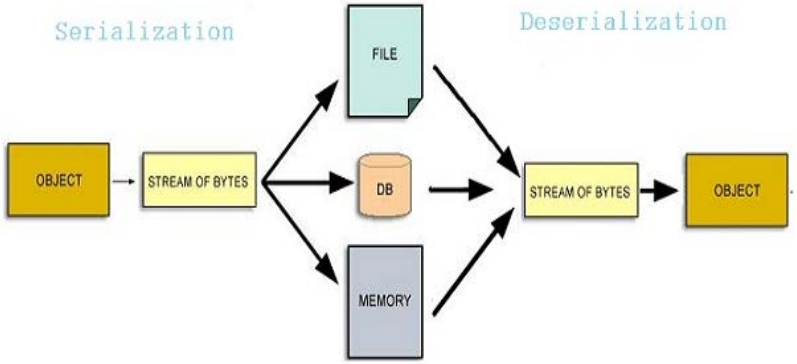
- Methods are not saved in the file. When we retrieve this data from a file by reading an object from the file, we retrieve the entire set of instance variables.

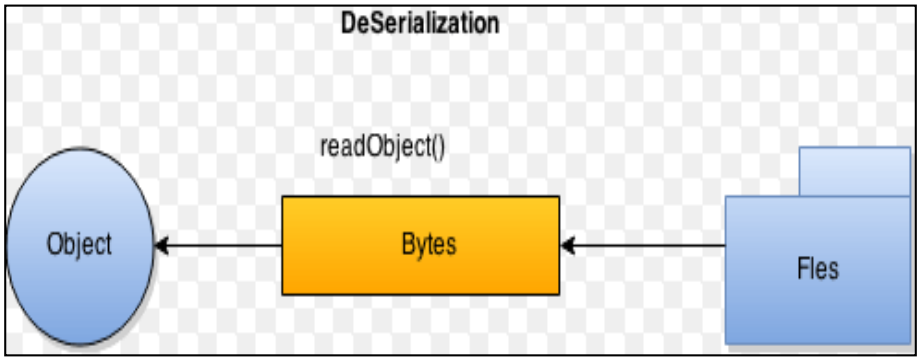
- Java provides a mechanism, called ***object serialization*** where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

Reading and Writing an object



To ***serialize*** an object means to convert its state to ***sequence of bytes .***





The reverse process of creating object from ***sequence of bytes*** is called ***deserialization***.

Object serialization

- After a serialized object has been written into a file, it can be read from the file and **deserialized** that is, 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 (like Windows) and deserialized on an entirely different platform (like Linux).

- Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object. In our course we use two methods: **writeObject** and **readObject**.

- The **writeObject** method serializes an Object and sends it to the output stream. Similarly, the **readObject** method retrieves the Object out of the stream and deserializes it.

Class Student serializabled

```
public class Student implements java.io.Serializable
 {
        private String fName;
        private String sName;
        private String address;
        public Student( String fName, String sName, String address)
        {
              this.fName = fName;
              this.sName = sName;
              this.address = address;
        } // constructor
   // rest methods
.
.
.
     public String toString()
     {
              String str = this.fName + "  “ + this.sName + "  “ + this.address;
              return str;
     } // toString
  } // Student
```

The class must implement the **java.io.Serializable interface**

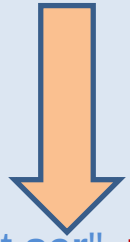
Serialization is the *conversion* of an object to a series of bytes, so that the object can be easily saved to persistent storage or streamed across a communication link.

The byte stream can then be **deserialised** – converted into a replica of the original object.

Writing an object to a file

```
public static void main(String[ ] args)  {
    Student[ ] writeArray = new Student[3];
    for(int i = 0; i < 3; i++)  {
        String fName = reader.next();
        String sName = reader.next();
        String address = reader.next();
        writeArray[i] = new Student(fName,sName,address);
    } // for
    try  {
          FileOutputStream   outFile = new FileOutputStream("e:/test.ser", true);
          ObjectOutputStream  out = new ObjectOutputStream(outFile);
          for(int j = 0; j < 3; j++)
                    out.writeObject(writeArray[j]);
          out.close();
          outFile.close();
    } // try
    catch(IOException e)  {
              System.out.println("Error I/O " + e);
    } //catch
} // main
```

Note: When serializing an object to a file, the standard convention in Java is to give the file a **.ser** extension.



Reading an object from a file

```
public static void main(String[ ] args)
 {
    Student[ ] readArray = new Student[3];
    try  {
          FileInputStream  inFile = new FileInputStream("e:/test.ser“);
          ObjectInputStream  in = new ObjectInputStream(intFile);
          for(int j = 0; j < 3; j++)
                readArray[i]  = (Student) in.readObject();
          in.close();
          inFile.close();
    } // try
    catch(IOException e)  {
                System.out.println(“Error I/O “ + e);
    } //catch
    System.out.println(“Deserializated student's data:" );
    for(int j = 0; j < 3; j++)
              System.out.println(readArray[j]);
} // main
```

Reading and Writing an object

Input data

```
Enter student first name-> David
Enter student second name-> Cohen
Enter student address-> MosheSharet11

Enter student first name-> Tal
Enter student second name-> Fridman
Enter student address-> Rager205

Enter student first name-> Ronit
Enter student second name-> Mizrahi
Enter student address-> BenGurion19
```

Output data

```
Deserializated student's data:

David Cohen MosheSharet11
Tal Fridman Rager205
Ronit Mizrahi BenGurion19
```