

Core Java

Session4 By Saurabh Sharma

Agenda

Today we will cover the following two modules:

- ❑ Exceptions and Error handling
- ❑ Input and Output Streams.

Session 4: Objectives

At the end of Session 4, you should be able to:

- ❑ Describe exceptions as well as error handling.
- ❑ Identify input and output Streams.

Module 1: Objectives

After completion of this module, you should be able to:

- ❑ Explain exceptions.
- ❑ Define uses, types, and categories of exceptions.
- ❑ Identify the various types of exceptions in Java .
- ❑ Implement exception handling using try, catch, throws, throw and finally clauses.
- ❑ Describe exception propagation and implement user-defined exceptions .

Defining Exception

An exception can be defined as an abnormal event that occurs during program execution and disrupts the normal flow of instructions.

Examples:

- ❑ Attempt to divide an integer by zero causes an exception to be thrown at run time.
- ❑ Attempt to call a method using a reference that is null.
- ❑ Attempting to open a nonexistent file for reading.
- ❑ JVM running out of memory.

Uses of exception handling

- ❑ To recover from the error conditions.
- ❑ To give users friendly, relevant messages when something goes wrong.
- ❑ To conduct certain critical tasks such as “save work” or “close open files/sockets” in case critical error leads to abnormal termination.
- ❑ To allow programs to terminate gracefully or operate in degraded mode.
- ❑ Exceptions don't always have to be errors, maybe your program can cope with the exception and keep going!
- ❑ Separating error handling code from Regular code.

Types of errors

- ❑ **Compile Time Error:** Occurs during compilation of a program which includes syntax errors, semantic errors etc.
- ❑ **Run Time Error:** Generated while running the program. For example, program that connects to another machine to fetch some data but found the target machine off, a error will encounter.

Exception occurrence reason

- ❑ Running out of memory.
- ❑ Resource allocation error.
- ❑ Inability to find files.
- ❑ Problem in network connectivity.
- ❑ Problem with database driver or connection.

Exception occurrence levels

- ❑ Hardware/operating system level.
 - ❖ Arithmetic exceptions; divide by 0, under/overflow.
 - ❖ Memory access violations; segment fault, stack over/underflow.
- ❑ Language level.
 - ❖ Type conversion; illegal values, improper casts.
 - ❖ Bounds violations; illegal array indices.
 - ❖ Bad references; null pointers.
- ❑ Program level.
 - ❖ User defined exceptions.

Types of exception

- ❑ There are three types of exception:
 - ❖ **Unchecked exceptions or runtime exception**
 - Compiler does not enforce code to be written to handle exception.
 - All unhandled (uncaught) unchecked exceptions are handled by JVM.
 - Exceptions that we encountered so far are all unchecked exceptions.
 - **Error** and all its subclasses are unchecked exception.
 - Subclass of **Exception** called **RuntimeException** and all its subclasses are unchecked exception.

Types of exception continued

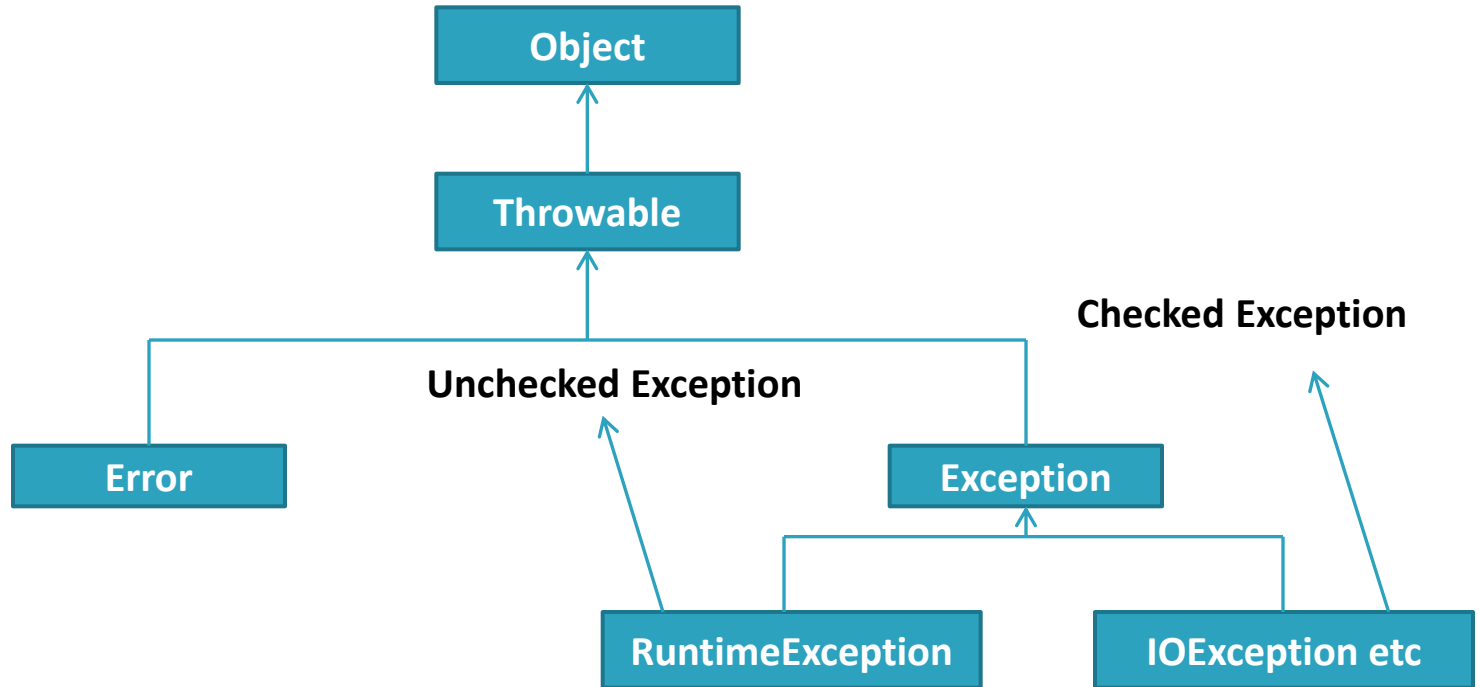
❑ **Checked exceptions or compiler-enforced exception.**

- ❖ Requires programmer to explicitly write code to handle exception otherwise compiler will throw an error.
- ❖ All the classes that do not inherit from **RuntimeException** are checked exception.
- ❖ Examples: **IOException**, **SQLException**, **CloneNotSupportedException**.

❑ **Error.**

- ❖ These exceptions are external to the application and application cannot anticipate errors like virtual memory errors, JVM errors etc.
- ❖ Java Documentation states that an **Error** is something “a reasonable application should not try to catch”.
- ❖ **Error** and its subclasses come under this.

Java exception hierarchy



Types of exception continued

❑ **RuntimeException / Unchecked Exception**

- ❖ ArithmeticException.
- ❖ NullPointerException.
- ❖ ArrayIndexOutOfBoundsException.
- ❖ ClassCastException.
- ❖ And many more classes.

❑ **Checked Exception**

- ❖ IOException.
- ❖ ClassNotFoundException.
- ❖ SQLException.
- ❖ IllegalAccessException etc.

Implementing exception handling

- ❑ try
- ❑ catch
- ❑ throws
- ❑ throw
- ❑ finally


Implementing exception handling

- ❑ Using try and catch statements:
 - ❖ The try block encloses the statements that might raise an exception within it and defines the scope of the exception handlers associated with it.
 - ❖ The catch block is used as an exception-handler. You enclose the code that you want to monitor inside a try block to handle a run time error.

- ❑ Syntax:

```
try{  
    ...  
}  
catch(ExceptionType1 e){ ...}  
catch(ExceptionType2 e){ ...}  
finally { ... }
```

Code that may throw exception



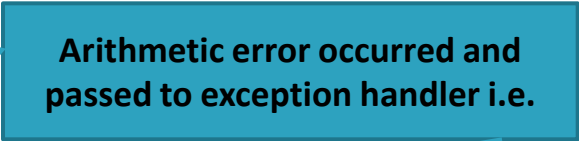
Collect your thoughts

```
public class UnitRate
{
    public void calculatePerUnitRate()
    {
        int qty=40, rate=0,punit=0;
        punit=qty/rate;
        System.out.println("The Per Unit Rate is = "+punit);
    }
}
```

- ❑ What will happen on execution of this code?
- ❑ It is important that we know what exceptions (particularly runtime exceptions) may be thrown by a piece of code so that handlers can be written.

Example: Exception handling in the code

```
public class UnitRate {  
    public void calculatePerUnitRate()  
    {  
        int qty=20, rate=0,punit=0;  
        try  
        {  
            punit=qty/rate;  
        }  
        catch(ArithmeticException e){  
            System.out.println("The product rate cannot be Zero, So Per Unit Rate Displayed Below is Invalid ");  
        }  
        System.out.println("The Per Unit Rate is = "+punit);  
    }  
}
```



Arithmetic error occurred and passed to exception handler i.e.

- ❑ This code handles ArithmeticException by providing useful information to the end user.
- ❑ Note that this helps the user to know what went wrong exactly.
- ❑ The error which was returned by JVM was not end-user friendly.

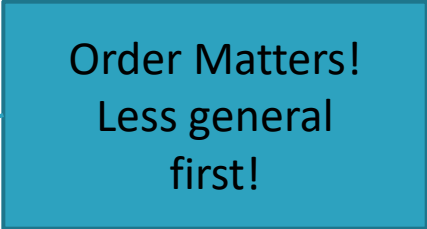
Flow after exception

- ❑ When exception occurs, the statements after exception are skipped and the control goes to the matching catch block. After the catch block, the control goes to the statement next to all the other catch blocks.
- ❑ Example catches an **ArithmeticException**. An **ArithmeticException** is a unchecked exception that is thrown when an attempt to divide by 0 is made.
- ❑ **Using multiple catch statement/blocks**
 - ❖ A single try block can have zero, one or many catch blocks.
 - ❖ The multiple catch blocks generate unreachable code error.
 - ❖ If the first catch block contains the Exception class object then the subsequent catch blocks are never executed. If this happens it is known as unreachable code problem.
 - ❖ To avoid unreachable code error, the last catch block in multiple catch blocks must contain the Exception class object.

Example: Using multiple catch blocks.

```
class Test
{
    public static void main(String as[])
    {
        try {
            // dangerous code here!
        }
        catch(ArithmeticException e) {
            // Specific error handling here
        }
        catch(RuntimeException e) {
            // More general error handling here
        }
    }
}
```

Order Matters!
Less general
first!



Exception class

- ❑ **Exception** object is checked exception. All exception classes are subtypes of the `java.lang.Exception` class.
- ❑ Constructors:
 - `public Exception()`**
 - `public Exception(String message)`**
- ❑ Important methods:
 - `public String getMessage()`**
 - `public void printStackTrace()`**
 - `StackTraceElement getStackTrace`**

Using finally block

- ❑ The finally block is used to process certain statements, no matter whether an exception is raised or not.
- ❑ Let us say we are performing some file operations. Invariably all IO methods in Java throws checked exception called **IOException**.
- ❑ Now if an exception occurs in between, we must make sure that files that we opened are closed before continuing further.

- ❑ Syntax:

```
try{  
    // Block of code  
}  
catch(Exception e){ ..... }  
finally{  
    // Block of code that is always executed irrespective of an exception being raised or not.  
}
```

Note: - finally can be without catch but try must be with either catch or finally or both.

Throwing an exception

- ❑ The throw statement causes termination of the normal flow of control of the Java code and stops the execution of the subsequent statements if an exception is thrown when the throw statement is executed.
- ❑ The throw clause transfers the control to the nearest catch block handling the type of exception object throws.
- ❑ The following syntax shows how to declare the throw statement:

- ❖ throw ThrowableObj

- ❖ Example:

```
public double divide(int dividend, int divisor) throws ArithmeticException {  
    if(divisor == 0) {  
        throw new ArithmeticException("Divide by 0 error");  
    }  
    return dividend / divisor;  
}
```

Throwing a checked exception

- ❑ What will happen with below code?

Example1:

```
public void setName(String name) {  
    if(name==null)  
        throw new Exception("Invalid name");  
    else this.name=name;    }
```

- ❑ A compilation error occurs:

Unreported exception java.lang.Exception; must be caught or declared to be thrown

Example2:

```
public void setName(String name) {  
    try{  
        if(name==null)  
            throw new Exception("Invalid name");  
        else    this.name=name;    }  
    }catch(Exception e){  
        System.out.println(e.getMessage());    }
```

Why do we need exception handlers

- ❑ We can achieve the same thing that the previous example does without writing any exception handlers.

```
public void setName(String name){  
    try { if(name==null)  
        System.out.println("Invalid Name");  
        else this.name=name;    }  
}
```

- ❑ But the reason for having exception handler is deeper than just a technical requirement. We need such handlers :
 - ❖ To separate normal business logic code and error handling code .
 - ❖ To take advantages of common error handlers.
 - ❖ To delegate.

Using throws exceptions

- ❑ The throws statement is used by a method to specify the types of exceptions the method throws.
- ❑ If a method is capable of raising an exception that it does not handle, the method must specify that the exception has to be handled by the calling method.
- ❑ This is done using the throws statement.
- ❑ The throws list can contain all the exception a method throws and does not want to handle.
- ❑ The exception classes in the exception list can be either exact match or automatically convertible to the exception object thrown by the code.
- ❑ Syntax:
<method declaration statement> throws <ExceptionList>{ ...}
- ❑ Example:
`public void add(int a, int b) throws ArithmeticException`

Example: throws exception

```
1. public abstract class Person{
2.     public void setName(String name)throws Exception{
3.         if(name==null)
4.             throw new Exception("Invalid name");
5.         else
6.             this.name=name;
7.     }
8. }
```

Note: - Any method that calls **setName()** methods must handle this exception.

Chained Exceptions using throws

- ❑ A large application may throw an exception whose root cause may be somewhere deep inside.
- ❑ The methods that handle the exception deep inside may partially handle the original exception and delegate the rest of the handling to the called method. This delegation can be done by either re-throwing-
 - ❖ the same exception object
 - ❖ totally new exception object
 - ❖ wrapped exception object (Exception Wrapping)
- ❑ In effect, this leads to one exception causing another exception and so on. Such type of exception occurrences are called Chained Exceptions.

Example:

```
1. public class Person
2. {
3.     public void setAge(int age) throws Exception
4.     {
5.         if(age<25)
6.             throws new Exception()
7.         else
8.             System.out.println("Age="+age);
9.     }
```

```
10. public static void main(String as[])
11. {
12.     Person p=new Person();
13.     try
14.     {
15.         p.setAge(20)
16.     }
17.     catch(Exception e)
18.     {
19.         System.out.println("Invalid Age");
20.     }
21. }
22. }
```

Output:- Invalid Age

- ❑ If age less than **25** is sent main method, an exception is thrown by setAge. This is partially handled by setAge and it is thrown once again to the main method. The main method provides another handler.

User defined exceptions: Creating your own exceptions

- ❑ Defining your own exceptions let you handle specific exceptions that are tailor-made for your application.
- ❑ Real world applications will have many errors that may occur and these errors will have to be classified such that handlers are written based on the type of error.
- ❑ So there is a need to create a new Exception class which will map to the application exceptions.
- ❑ Only objects of **Throwable** class can be thrown.
- ❑ So new (user-defined classes) can either inherit from **Throwable** class or more correctly inherit from **Exception** class (because subclasses of **Exception** classes are supposed to define application exception).

Example: User defined exceptions

```
1. class MyException extends Exception
2. {
3.     public String message(){
4.         return "Invalid Age";
5.     }
6. }
7. public class Person
8. {
9.     public void setAge(int age) throws MyException
10.    {
11.        if(age<25)
12.            throws new MyException()
13.
14.        else
15.            System.out.println("Age="+age);
16.    }
```

```
17.     public static void main(String as[])
18.     {
19.         Person p=new Person();
20.         try
21.         {
22.             p.setAge(20)
23.         }
24.         catch(MyException e)
25.         {
26.             System.out.println(e.message());
27.         }
29.     }
30. }
```

Output:- Invalid Age

Automatic resource management with try-with-resources

- ❑ This is the new feature of java 7.
- ❑ This is a try statement that declares one or more resources. A resource is as an object that must be closed after the program is finished with it.
- ❑ The **try-with-resources** statement ensures that each resource is closed automatically at the end of the statement. Any object that implements the `java.lang.AutoCloseable` or `java.io.Closeable` interface can be used as a resource.
- ❑ Prior to **try-with-resources** (before Java 7) while dealing with SQL Statement or ResultSet or Connection objects or other IO objects one had to explicitly close the resource in finally block. We discuss about ResultSet, Connection and IO objects in next coming session.

Example: try-with-resources

❑ Instead of

```
public void oldTry() {  
    try {  
        fos = new FileOutputStream("movies.txt");  
        dos = new DataOutputStream(fos);  
        dos.writeUTF("Java 7 Block Buster");  
    } catch (IOException e) {  
        e.printStackTrace();  
    } finally {  
        try {  
            fos.close();  
            dos.close();  
        } catch (IOException e) {  
            // log the exception  
        }  
    }  
}
```

❑ You can now say:

```
public void newTry()  
{  
    try (FileOutputStream fos = new FileOutputStream("movies.txt");  
        DataOutputStream dos = new DataOutputStream(fos))  
    {  
        dos.writeUTF("Java 7 Block Buster");  
    }  
    catch (IOException e)  
    {  
        // log the exception  
    }  
}
```


Multi-catch

- ❑ This is also a new feature of java 7.
- ❑ You'll now be able to catch multiple exceptions type in one catch block.
- ❑ Example:

```
public void newMultiMultiCatch() {  
    try {  
        methodThatThrowsThreeExceptions();  
    } catch (ExceptionOne e) {  
        // deal with ExceptionOne  
    } catch (ExceptionTwo | ExceptionThree e) {  
        // deal with ExceptionTwo and ExceptionThree  
    }  
}
```

- ❑ Notice how the two exception class names in the second catch block are separated by the pipe character |. The pipe character between exception class names is how you declare multiple exceptions to be caught by the same catch clause.

Overriding and Exception

- ❑ An overridden method CANNOT throw :

- ❖ new checked exceptions.
- ❖ parent class exception.

- ❑ An overridden method

- ❖ can throw child class exception.
- ❖ completely omit the exception.

- ❑ Example:

```
class Test {  
    public boolean equals(Object obj) throws Exception  
    {  
        return true;  
    }  
}
```

Assertion

- ❑ An assertion is a statement in the Java™ programming language that enables test the assumptions about program.
- ❑ Each assertion contains a boolean expression that is assumed to be true. When executed if the boolean expression is false, the system will throw an error.
- ❑ Writing assertions while programming is one of the quickest and most effective ways to detect and correct bugs. As an added benefit, assertions serve as internal documentation, thus enhancing maintainability.
- ❑ These are created with manual unit testing and debugging in mind.
- ❑ This facility is very similar to the C/C++ assertion.
- ❑ Syntax : 2 forms
 - ❖ **assert (condition);**
 - ❖ **assert(condition): statement returning a string;**
- ❑ If the condition is **false**, a runtime exception called **AssertionError** is thrown (code executes in assertion enabled mode)

Assertion enabling/disabling

- ❑ In order for the javac compiler to accept code containing assertions, you must compile the code with the following.
- ❑ `javac -source 1.4 MyClass.java` .
- ❑ To enable assertions use `-enableassertions`, or `-ea`, option at runtime. To disable assertions use `-disableassertions`, or `-da`. Specify the granularity with the arguments provided at runtime.
 - ❖ no arguments - Enables or disables assertions in all classes except system classes.
 - ❖ `packageName` - Enables or disables assertions in the named package and any subpackages.
 - ❖ `...` - Enables or disables assertions in the unnamed package in the current working directory.
 - ❖ `className` - Enables or disables assertions in the named class.
- ❑ In Eclipse, this can be done in “**x=Arguments**” tab of “**Run Configuration**”. In **VM Arguments** section enter **-ea**.

Example: Assertion

```
1. import java.util.*;
2. import java.util.Scanner;
3. public class AssertionExample {
4.     public static void main( String args[] ) {
5.         Scanner scanner = new Scanner( System.in );
6.         System.out.print( "Enter a number between 0 and 20: " );
7.         int value = scanner.nextInt();
8.         assert( value >= 0 && value <= 20 ) : "Invalid number: " + value;
9.         System.out.println( "You have entered "+value );
10.    }
11. }
```

Example: Assertion continued

- ❑ In the previous example, When the user enters the number `scanner.nextInt()` method reads the number from the command line. The `assert` statement determines whether the entered number is within the valid range. If the user entered a number which is out of range then the error occurs.
- ❑ To run the above example,
Compile the example with: `javac AssertionExample.java`
Run the example with: `java -ea AssertionExample`
To enable assertions at runtime, `-ea` command-line option is used
- ❑ When you enter the number within range, output will be displayed as:

```
E:\>java -ea AssertionExample  
Enter a number between 0 and 20: 6  
You have entered 6
```

Example: Assertion continued

- ❑ When you enter the number out of range, output will be:

```
E:\>java -ea AssertionExample
```

```
Enter a number between 0 and 20: 45
```

```
Exception in thread "main" java.lang.AssertionError: Invalid number: 45
```

```
at AssertionExample.main(AssertionExample.java:8)
```

- ❑ AssertionError:
 - ❖ This is a class that inherits from Error in java.lang package.
 - ❖ This error is thrown at runtime when assert condition fails.
 - ❖ This error must be seen only if code is executed with assertion enabled.
 - ❖ So while you are free to throw an **AssertionError** from the code, this is not the desired usage.



Core Java: Session 4

Module 2: Input and Output Streams

Module 2: Objectives

After completion of this module, you should be able to:

- ❑ Explain Input output streams and its different uses
- ❑ Explain and serialize the objects

File class

- ❑ **java.io.File** class can be used to work with system dependent commands for files and directories.
- ❑ The path name in the code hence will depend on the underlying OS in which JVM is installed.
- ❑ To make the code portable so that it works on all systems, **static** member **separator** defined in the **File** class can be used.
- ❑ The path name can be either *absolute* or *relative*.
- ❑ Constructors:
 - ❖ **File(String pathname)**
 - ✓ Creates a new **File** instance (if relative path is given then it converts it into abstract pathname)
 - ❖ **File(String parent, String child)**
 - ❖ **File(File parent, String child)**
 - ✓ Creates a new **File** instance from a parent pathname and a child pathname string. If “parent” is **null** then it behaves like the single-argument File constructor

Members in File class

- ❑ **boolean createNewFile() throws IOException**
 - ❖ creates a new, empty file if a file with the name (as specified in the constructor) does not exist and returns true; otherwise it returns false. Note that this method throws **IOException** if I/O error occurs.
- ❑ So to create a file:
 - ❖ Create instance of **File** object.
 - ❖ Call **createNewFile** method.
- ❑ **static final String separator**
 - ❖ system-dependent separator character.
- ❑ **boolean delete()**
 - ❖ Deletes the file or directory. Directory must be empty in order to be deleted. Returns **true** if the delete operation is successful.
- ❑ **boolean mkdir()**
 - ❖ Creates the directory named by the pathname. Returns true if the directory was created; false otherwise .

Members in File class continued

- ❑ **boolean mkdirs()**
 - ❖ Creates the directory named by this pathname, including any necessary but nonexistent parent directories. Note that if this operation fails it may have succeeded in creating some of the necessary parent directories.
- ❑ **void deleteOnExit()**
 - ❖ Deletes the file or directory when the virtual machine terminates. Deletion happen only for normal termination of the virtual machine. Once deletion has been requested, it is not possible to cancel the request.
- ❑ **boolean renameTo(File dest)**
 - ❖ Renames the file. This is system dependent so return value should always be checked to make sure that the rename operation was successful.
- ❑ **boolean isDirectory()**
 - ❖ Returns true if the File object denotes a directory; false otherwise.
- ❑ **String getName()**
 - ❖ Returns the name of the file or directory . just the last name in the pathname's name sequence.

Members in File class continued

- ❑ **String[] list()**
 - ❖ Returns list names files and directories in the directory.
- ❑ **boolean isFile()**
 - ❖ Returns true if the File object denotes a file; false otherwise.
- ❑ **boolean isHidden()**
 - ❖ Returns true if the File object is a hidden file; false otherwise.
- ❑ **boolean exists()**
 - ❖ Returns true if the file or directory exists; false otherwise.
- ❑ **String getAbsolutePath()**
 - ❖ Returns the absolute pathname string of this abstract pathname.
- ❑ **long lastModified()**
 - ❖ Returns the time that the file object was last modified.
- ❑ **long length()**
 - ❖ Returns the length of the file, unspecified it is a directory.

Members in File class continued

- ❑ **boolean canExecute()**
- ❑ **boolean canRead()**
- ❑ **boolean canWrite()**
 - ❖ Returns true if the application can execute/read/write the file denoted; false otherwise.
- ❑ **boolean setXxx(boolean permission)**
- ❑ **setXxx(boolean permission, boolean ownerOnly)**
 - ❖ Xxx could be **Readable, Executable or Writable**.
 - ❖ Sets the read/execute/write permission if **permission** is true.
 - ❖ **ownerOnly** is true then the read/execute/write permission applies only to the owner's execute permission provided underlying file system can distinguish between owner and others; otherwise, it applies to everybody.

Example: Creating a file

❑ Creates a new file named **myFile.txt**. If file exists then it deletes the file and creates a new one.

```
1. import java.io.*;
2. class TestFile{
3.     public static void main(String str[]){
4.         try{
5.             File file = new File("myFile.txt");
6.             if(file.exists())
7.                 file.delete();
8.             boolean b=file.createNewFile();
9.             System.out.println(b);
10.        }catch(IOException e){ }
11.    }}
```

Overview of input output (IO)

- ❑ Many Java programs need to read or write data. This can be obtained by using the classes in `java.io` package.
- ❑ Your program can get input from a data source by reading a sequence characters from a `InputStream` attached to the source.
- ❑ Your program can produce output by writing a sequence of characters to an `OutputStream` attached to a destination.

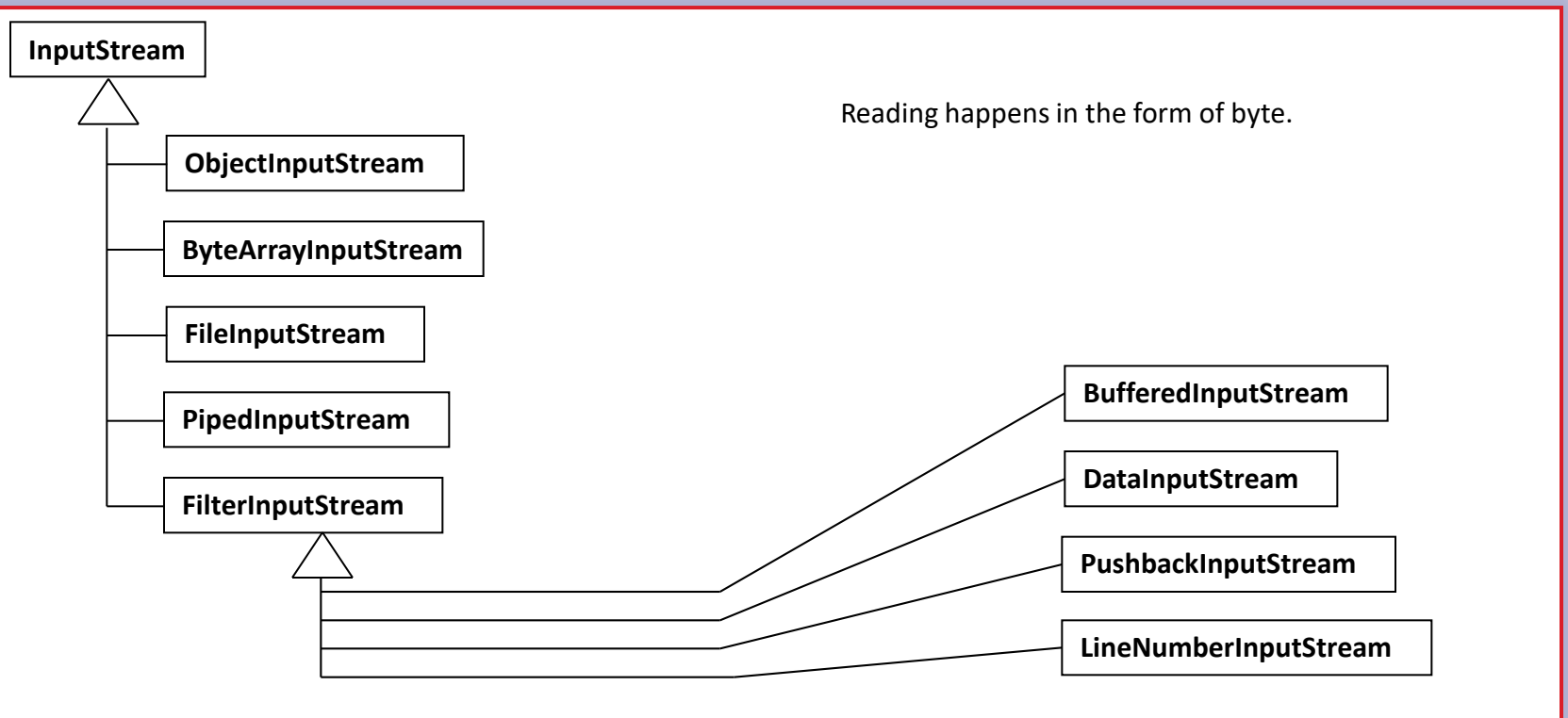
What are streams?

- ❑ A stream is a flowing set of data.
- ❑ Streams support many different kinds of data, including simple bytes, primitive data types.
- ❑ Some streams simply pass on data, others manipulate and transform the data in useful ways.
- ❑ There are 2 types of stream:
 - ❖ Input stream to read data from a source. An input stream may be files, keyboard, console, other programs, a network, or an array!
 - ❖ Output stream to read data into a destination. An output stream may be disk files, monitor, a network, other programs, or an array.
- ❑ Fundamentally stream may be :
 - ❖ Byte stream : data read or written is in the form of byte.
 - or
 - ❖ Character stream: data read or written is in the form of character.

InputStream

- ❑ A program uses an input stream to read data from source, one item at a time. Source can be File, Socket, Array or Thread.
- ❑ The InputStream class is an abstract superclass that provides a minimal programming interface and a partial implementation of input streams.
- ❑ The InputStream class defines methods for reading bytes or arrays of bytes.

Hierarchy of Input Stream



InputStream Continued

❑ FileInputStream:

- ❖ Read data from file on the native file system.
- ❖ Is meant for reading streams of raw bytes such as image data.
- ❖ Constructors frequently used:
 - ✓ `FileInputStream (String filename);`
 - ✓ `FileInputStream(File file);`

❑ ByteArrayInputStream

- ❖ A `ByteArrayInputStream` contains an internal buffer that contains bytes that may be read from the stream.
- ❖ An internal counter keeps track of the next byte to be supplied by the read method.
- ❖ Commonly used Constructors
 - ✓ `public ByteArrayInputStream(byte[] buf)`
 - ✓ `public ByteArrayInputStream(byte[] buf, int offset, int length)`

InputStream Continued

- ❑ `SequenceInputStream` :
 - ❖ Concatenate multiple input streams into one input stream.
- ❑ `StringBufferInputStream` :
 - ❖ Allow programs to read from a `StringBuffer` as if it were an input stream.
- ❑ `PipedInputStream`:
 - ❖ Data can be read from a thread by using `PipedInputStream`.
- ❑ `FilterInputStream`:
 - ❖ `FilterInputStream` is subclasses of `InputStream`.
 - ❖ These classes define the interface for filtered streams which process data as it's being read or written.

InputStream Continued

❑ BufferedInputStream

- ❖ Buffered input streams read data from a memory area known as a buffer; the native input API is called only when the buffer is empty.
- ❖ Constructor commonly used:
 - ✓ `BufferedInputStream(InputStream in)`

❑ DataInputStream

- ❖ A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way.
- ❖ Constructor used
 - ✓ `DataInputStream(InputStream in)`

Frequently used Methods of InputStream

- ❑ `public int read()` throws `IOException`
 - ❖ Reads a byte of data from this input stream.
- ❑ `public int read(byte [] b)` throws `IOException`
 - ❖ Reads up to `b.length` bytes of data from this input stream into an array of bytes .
- ❑ `public int read(byte[] b, int off, int len)` throws `IOException`
 - ❖ Reads bytes from this byte-input stream into the specified byte array, starting at the given offset.
- ❑ `public long skip(long n)` throws `IOException`
 - ❖ Skips over and discards `n` bytes of data from the input stream.
- ❑ `public void close()` throws `IOException`
 - ❖ Closes this file input stream and releases any system resources associated with the stream.

Frequently used Methods of DataInputStream

- ❑ `public final byte readByte() throws IOException`
 - ❖ Reads and returns one input byte.
- ❑ `public final float readFloat() throws IOException`
 - ❖ Reads four input bytes and returns a float value.
- ❑ `public final double readDouble() throws IOException`
 - ❖ Reads eight input bytes and returns a double value.
- ❑ `public final char readChar() throws IOException`
 - ❖ Reads an input char and returns the char value.
- ❑ `public final boolean readBoolean() throws IOException`
 - ❖ Reads one input byte and returns true if that byte is nonzero, false if that byte is zero.

Example: Reading data from file

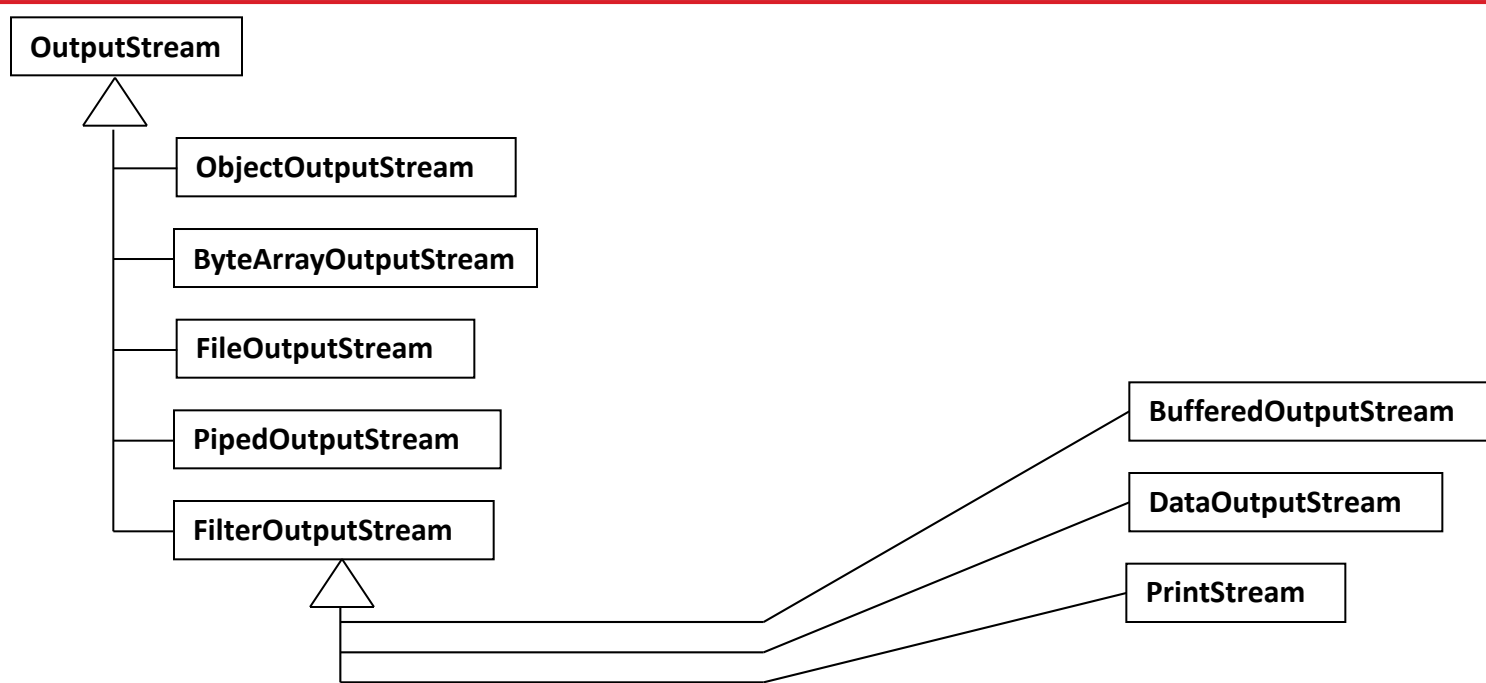
```
1. import java.io.*;
2. public class ReadFileExample
3. {
4.     FileInputStream fis;
5.     BufferedInputStream bis;
6.     ReadFileExample() throws IOException
7.     {
8.         fis=new FileInputStream("test.txt");
9.         bis=new BufferedInputStream(fis);
10.    }
11.    public void readData()throws IOException
12.    {
13.        int no=bis.read();
14.        while(no!=-1)
15.        {
16.            char val=(char)no;
17.            no=bis.read();
```

```
18.        if(val=='\n')
19.            System.out.println();
20.        else
21.            System.out.print(val);
22.        }
23.    }
24.    public static void main(String s[])throws
        IOException
25.    {
26.        ReadFileExample f=new ReadFileExample();
27.        f.readData();
28.    }
29. }
```

OutputStream

- ❑ A program uses an output stream to write data to a destination, one item at time: Destination can be File, Socket, Array or Thread.
- ❑ The OutputStream class is an abstract superclass that provides a minimal programming interface and a partial implementation of output streams.
- ❑ OutputStream defines methods for writing bytes or arrays of bytes.

Hierarchy of OutputStream



Commonly used methods of OutputStream

- ❑ `public void write(byte[] b) throws IOException`
 - ❖ Writes `b.length` bytes from the specified byte array to this file output stream.
- ❑ `public void write(int b) throws IOException`
 - ❖ Writes the specified byte to this file output stream.
- ❑ `public void flush() throws IOException`
 - ❖ Flushes the stream.
- ❑ `public void close() throws IOException`
 - ❖ Closes this file output stream and releases any system resources associated with this stream. This file output stream may no longer be used for writing bytes.

PrintStream

- ❑ **PrintStream** is a class that has functionality like the ability to print representations of various data values conveniently.
- ❑ **System.out** is an instance of **PrintStream**.
- ❑ Apart from this, two other functionalities that are provided here are:
 - ❖ Unlike other output streams, a **PrintStream** never throws an **IOException**.
 - ❖ The **flush()** method can be made to automatically invoked after **println** method is invoked or newline (**'\n'**) is written.
- ❑ Constructor
 - ❖ `PrintStream(OutputStream out)`
 - ❖ `PrintStream(OutputStream out, boolean autoflush)`

Methods commonly used in PrintStream

- ❑ `void print(xxx b)`
- ❑ `void println(xxx b)`
 - ❖ where xxx is any primitive type, String or Object.
- ❑ `public void close()`
 - ❖ Closes the stream.
- ❑ `PrintStream printf(String format, Object... args)`
- ❑ `PrintStream format(String format, Object... args)`
 - ❖ Both of the above methods have same functionality.
 - ❖ We have been using theses method extensively through System.out.

Object Serialization

- ❑ The mechanism of storing the state of an object in the hard disk so that it can be restored later by your program.
- ❑ Serialization enables storing values of all instance variables which includes both primitives and **Serializable** objects.
- ❑ Serialization mechanism creates a file into which the state of the object is written.
- ❑ This file can later be read by the java program which can then restore the object's state.
- ❑ When an object is serialized, its state and other attributes are converted into an ordered series of bytes.
- ❑ This byte series can be written into streams.
- ❑ **ObjectOutputStream** and **ObjectInputStream** classes are used for these purposes. They are wrapper classes that take **OutputStream** and **InputStream** objects respectively.

java.io.Serializable

- ❑ Only the objects which implement **Serializable** interface can be serialized.

class MyObject implements Serializable{... }

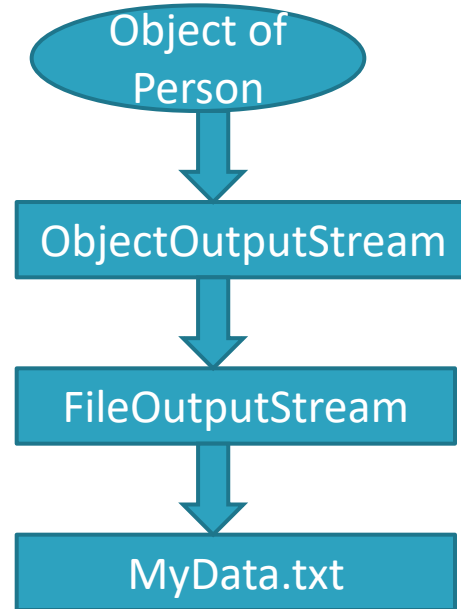
- ❑ **Serializable** is a marker interface.
- ❑ If object has references, then the references also must be either **Serializable** or should be marked **transient**.
- ❑ In JSE, some classes are not **Serializable**. For example **Thread class, Subclasses of Writer, Reader, InputStream, OutputStream**.
- ❑ All the collection classes, all primitive wrappers, **String, StringBuffer, StringBuilder** are **Serializable**.
- ❑ If an attempt to serialize an object that does not implement **Serializable** is made, **NotSerializableException** is thrown.

Classes used for Reading and writing objects

- ❑ The `ObjectInputStream` class:
 - ❖ **`ObjectInputStream(InputStream in)` throws `IOException`**
 - ❖ **`xxx readXxx()` where `xxx` is any primitive type, or **`Object`**.**
 - ❖ **`int read()`**
 - ❖ And all the methods from **`InputStream`**.
- ❑ The `ObjectOutputStream` class
 - ❖ **`ObjectOutputStream(OutputStream out)` throws `IOException`**
 - ❖ **`void writeXxx(xxx v)` where `xxx` is any primitive type, or **`Object`**.**
 - ❖ **`void write(int x)`**
 - ❖ And all the methods from **`OutputStream`**.

Procedure of writing Objects into File

```
import java.io.*;  
class Person implements Serializable  
{   String name;  
    int age;  
    --method for entering data----  
    ---main method----  
}
```




Example: Writing Object(Serialization)

```
1. import java.io.*;
2. public class Person implements Serializable {
3.   String name;
4.   int age;
5.   public Person(String name,int age)
6.   {
7.     this.name=name;
8.     this.age=age;
9.   }
10.}

11. class WritePerson
12. {
13.   public static void main(String as[])
14.   {
15.     Person p=new Person("Saurabh", 31);

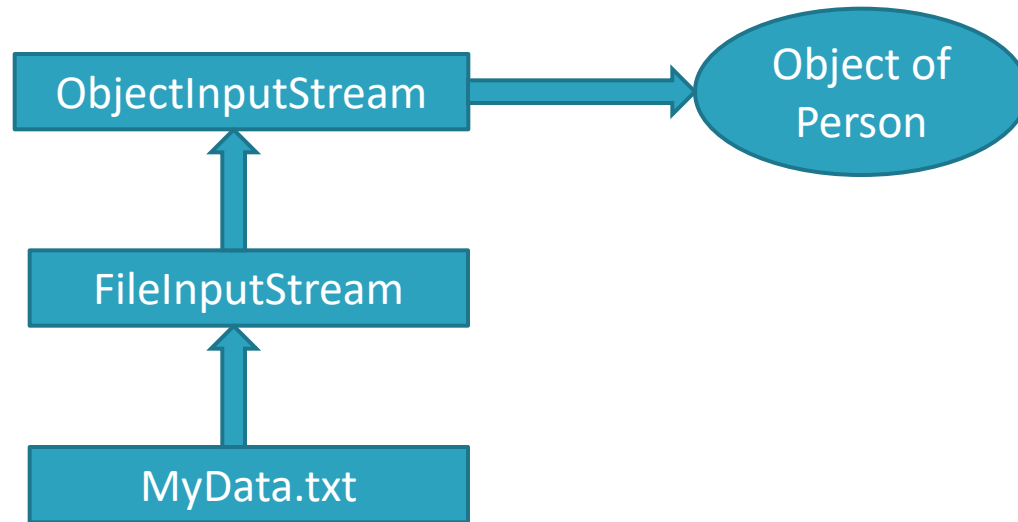
16.   try
17.   {
18.     FileOutputStream fos=new FileOutputStream("MyData.txt");
19.     ObjectOutputStream os=new ObjectOutputStream(fos);

20.     os.writeObject(p);
21.     os.flush();
22.     os.close();
23.   }
24.   catch(Exception e)
25.   {
26.     System.out.println("Error→"+e);
27.   }
28. }
29. }
```



Object of Person with name and age
written to MyData.txt

Procedure of reading Object from a File



Example: Reading Object(Deserialization)

```
1. import java.io.*;
2. class ReadPerson
3. {
4.     public static void main(String as[])
5.     {
6.         try
7.         {
8.             FileInputStream fis=new FileInputStream("MyData.txt");
9.             ObjectInputStream ois=new ObjectInputStream(fis);
10.
11.             Object obj=ois.readObject(ois);
12.             Person p=(Person)obj;
13.             System.out.println("Name="+p.name+" , Age="+p.age);
14.         }
15.         catch(Exception e)
16.         {
17.             System.out.println("Error→"+e);
18.         }
19. }}
```

Beware!

- ❑ You could save any number of object in a file.
- ❑ Objects are read back in the same sequence as they are written.
- ❑ Care must be taken while de-serializing the objects.
 - ❖ The objects must be cast into its correct type otherwise an **ClassCastException** will be thrown at runtime.
 - ❖ The objects must be retrieved in the same way as they are saved. For instance, if you save an integer using **writeInt()** then you must retrieve using **readInt()** method. Using **readObject()** and casting it back to **int** will not work(an **java.io.OptionalDataException** will be thrown at runtime).
- ❑ Safest and more common way to save and retrieved is to use **writeObject()** and **readObject()** methods.

transient

- ❑ Instance variables marked **transient** will not be saved.
- ❑ When object is de-serialized the **transient** variables are set to the default value based on their type.
- ❑ During serialization even the **private** state of the object is stored.
- ❑ Hence sensitive information like credit card number, password, a file descriptor contains a handle that provides access to an operating system resource must be marked transient.
- ❑ Also if a class contains references of object that cannot be serialized (like Thread), must be marked **Serializable** .
- ❑ Example:

```
class Person implements Serializable {  
String name;  
transient int age;  
---constructor to enterdata }
```



Value will not be persisted

Console IO

- ❑ Java 6 introduces a new class called **Console** in **java.io** package.
- ❑ This class has convenient method that can prompt the user for the input and read input from the terminal at the same time.
- ❑ It is also does has features that will not echo the password entry on the screen.
- ❑ This class is character based.
- ❑ **Console** is a **final** class with no public constructors. To obtain **Console** object:
System.console() method is used which returns the only instance that JVM has for this class.
- ❑ Read and write operations are **synchronized** methods.

JVM and Console

- ❑ JVM has a console that is dependent upon the underlying platform. This object can be obtained using the `System.console()` method that returns `Console` object.
- ❑ If the JVM is started from command line then its console will exist.
- ❑ If the virtual machine is started automatically or from an IDE then console will not be available.
- ❑ Therefore before using the `Console` object, a check for null has to be done to ensure that the object exists.

```
if ((cons = System.console()) != null){
```

```
...
```

```
}
```

Console Method

- ❑ **Console format(String fmt, Object... args)**
- ❑ **Console printf(String fmt, Object... args)**
 - ❖ Used to write formatted data. The fmt represents format string which are same as the one that was used for **System.out.printf**.
- ❑ **String readLine(String fmt, Object... args0)**
 - ❖ Prompts and reads a single line of text.
- ❑ **String readLine()**
 - ❖ reads a single line of text.
- ❑ **char[] readPassword(String fmt, Object... args)**
 - ❖ Prompts and reads a password or passphrase from the console with echoing disabled.
- ❑ **char[] readPassword()**
 - ❖ reads a password or passphrase from the console with echoing disabled.

Example:

```
1. import java.io.Console;
2. import java.util.Arrays;

3. public class Test {
4.     public static void main(String[] args) {
5.         String pass="abcd";
6.         Console c = System.console();
7.         if (c == null) {
8.             System.err.println("Console Object is not available.");
9.             System.exit(1); }
10.        String login = c.readLine("Login:");
11.        char [] pwd = c.readPassword("Password: ");
12.        String s=new String(pwd);
13.        if(s.equals(pass)) System.out.println("Right pwd");
14.        else System.out.println("Incorrect pwd");
15.    }}
```

Note that this code throws an exception at runtime when run on eclipse. It prints **“Console Object is not available”**. But it works fine when you run from the command prompt.

Reader and Writer Classes

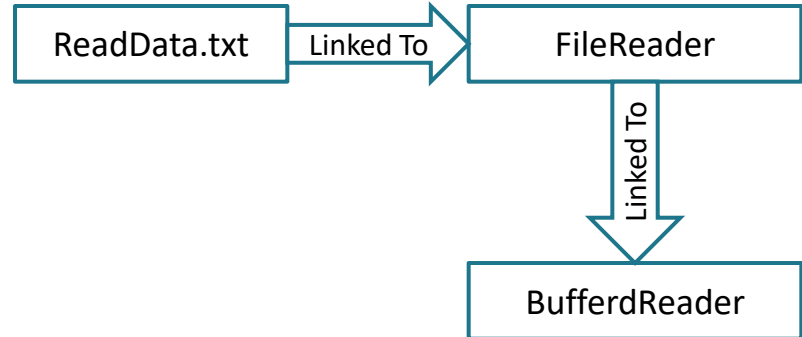
- ❑ Input and output for characters streams are handled through subclasses of the **Reader** and **Writer** classes.
- ❑ These classes are abstractions that Java provides for dealing with reading and writing character data.
- ❑ FileReader and FileWriter
 - ❖ Read data from or write data to a file on the native file system.
- ❑ PipedReader and PipedWriter
 - ❖ Pipes are used to channel the output from one program (or thread) into the input of another.
- ❑ CharArrayReader and CharArrayWriter
 - ❖ Read data from or write data to a char array in memory.

Filter Reader and Writers

- ❑ FilterReader and FilterWriters are subclasses of Reader and Writer, respectively. These classes define the interface for filtered streams which process data as it's being read or written.
- ❑ BufferedReader and BufferedWriter
 - ❖ Buffer data while reading or writing, thereby reducing the number of accesses required on the original data source.
- ❑ PrintWriter
 - ❖ An output stream with convenient print methods.

Example:

```
1. import java.io
2. class ReadPerson {
3.     public static void main(String as[]) {
4.         try {
5.             FileReader fr=new FileReader("ReadData.txt");
6.             BufferedReader br=new BufferedReader(fr);
7.             String str="";
8.             while((str=br.readLine()) !=null)
9.                 System.out.println(str);
10.        }
11.        catch(Exception e) { System.out.println("Error→"+e); }
12.    }
13. }
```



Introduction to NIO

- ❑ New I/O, usually called **NIO**, is a collection of APIs that offer additional capabilities for intensive I/O operations. It was introduced with the Java 1.4 release by Sun Microsystems to complement an existing standard I/O. The extended NIO that offers further new file system APIs, called NIO.2, was released with Java SE 7 (“Dolphin”).
- ❑ Java NIO offers a different way of working with IO than the standard IO API's.
 - ❖ **Java NIO: Channels and Buffers:** In the standard IO API you work with byte streams and character streams. In NIO you work with channels and buffers. Data is always read from a channel into a buffer, or written from a buffer to a channel.
 - ❖ **Java NIO: Asynchronous IO:** Java NIO enables you to do asynchronous IO. For instance, a thread can ask a channel to read data into a buffer. While the channel reads data into the buffer, the thread can do something else. Once data is read into the buffer, the thread can then continue processing it. The same is true for writing data to channels.
 - ❖ **Java NIO: Selectors:** Java NIO contains the concept of "selectors". A selector is an object that can monitor multiple channels for events (like: connection opened, data arrived etc.). Thus, a single thread can monitor multiple channels for data.

Why NIO2

- ❑ Since its earlier versions, Java was already capable of manipulating files. However, until Java 6 there were several limitations with the API provided by the JVM in package **java.io**, for instance:
 - ❖ There is no single operation to copy a file to another location. When necessary, it has to be implemented using data flows;
 - ❖ Support to file attributes (creation date, last update date, permissions to read/write, executable flag, etc.) is limited;
 - ❖ It's not 100% consistent throughout the different platforms supported by Java;
 - ❖ Often, exceptions thrown due to failures during file operations are not very useful;
 - ❖ It's not possible to extend the API to support a new type of file system.
- ❑ To address these limitations, More New I/O APIs for the Java™ Platform (“NIO.2”) was proposed. This extension was called “NIO.2” because since Java 1.4 the package **java.nio** is already present, supporting the creation of I/O channels. Channels are objects that represent a connection with a generic resource that is capable of executing I/O operations, such as files and network sockets.

NIO2 Continued

- ❑ The NIO.2 API adds three subpackages to **java.nio**:
 - ❖ **java.nio.file**: main package for NIO.2, defines interfaces and classes to access files and file systems using the new API.
 - ❖ **java.nio.file.attribute**: contains classes and interfaces related to reading and changing file attributes.
 - ❖ **java.nio.file.spi**: contains service provider interfaces for NIO.2, to be used by developers who wish to implement support to a new type of file system.

NIO2 Continued

- ❑ The following classes (from package **java.nio.file**) represent the main concepts of the new API:
 - ❖ **Path**: immutable class that represents the path to any file, like, for instance,
C:\Users\admin\My Documents\document.doc;
 - ❖ **Files**: class that contains several **static** methods for the execution of operations in files given their paths;
 - ❖ **FileSystem**: class that represents the file system as a whole, used to obtain paths to files;
 - ❖ **FileSystems**: class that contains several **static** methods for obtaining a file system. The method **FileSystems.getDefault()** obtains an object that allows us to access all files to which the JVM has access;
 - ❖ **FileStore**: class that represents the file storing mechanism that is manipulated by the different methods of the API (a partition of a hard drive, an external device plugged in via USB, etc.).

Example:

```
import java.nio.file.*;
public class PathInfo1
{
    public static void main(String[] args)
    {
        Path p = Paths.get("D:\\test\\testfile.txt");
        System.out.println("Printing file information: ");
        System.out.println("\t file name: " + p.getFileName());
        System.out.println("\t root of the path: " + p.getRoot());
        System.out.println("\t parent of the target: " + p.getParent());
        System.out.println("Printing elements of the path: ");
        for(Path element : testFilePath) {
            System.out.println("\t path element: " + element);
        }
    }
}
```

Output:

Printing file information:
file name: testfile.txt
root of the path: D:\
parent of the target: D:\test
Printing elements of the path:
path element: test
path element: testfile.txt

Example: Copying a File using Files.copy()

```
import java.io.IOException;
import java.nio.file.*;
public class FileCopy {
    public static void main(String[] args) {
        if(args.length != 2){
            System.out.println("usage: FileCopy <source-path>
<destination-path>");
            System.exit(1);
        }
        Path pathSource = Paths.get(args[0]);
        Path pathDestination = Paths.get(args[1]);
        try {
            Files.copy(pathSource, pathDestination);
            System.out.println("Source file copied successfully");
        } catch (IOException e)
        { e.printStackTrace(); }
    }
}
```

Let's execute it and see whether it works.

```
D:\> java FileCopy FileCopy.java Backup.java
Source file copied successfully
```

Questions

