# JAVA PROGRAMMING BASICS

Module 3: Java Standard Edition

## Training program

1. Java I/O Streams
2. Java Serialization
3. Java Database Connectivity
4. Java GUI Programming
5. The basics of Java class loaders
6. Reflections
7. Annotations
8. The proxy classes
9. Java Software Development
10. Garbage Collection

## Module contents

- Java I/O Streams
  - The concept of input-output  streams
  - Byte streams and character streams
  - The main I/O stream classes
  - The RandomAccessFile class
  - Principles of handling IO errors
  - A try-catch with resources
  - The "File" class. File operations
  - NIO.2

## Module contents

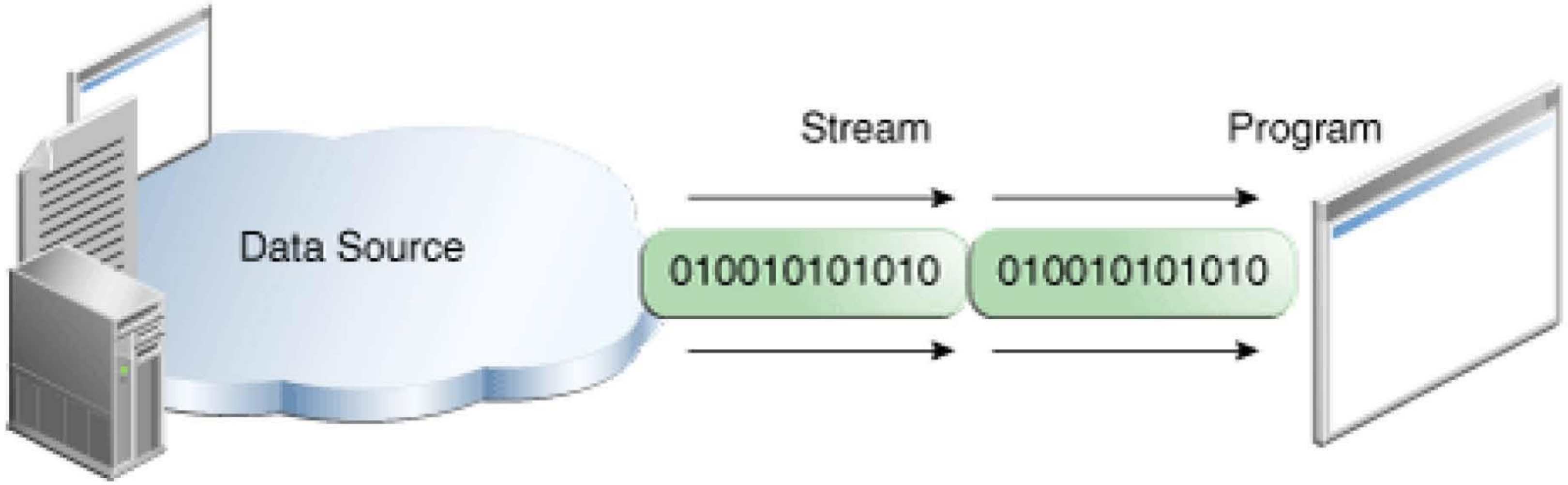
- Java I/O Streams
  - The concept of input-output  streams
  - Byte streams and character streams
  - The main I/O stream classes
  - The RandomAccessFile class
  - Principles of handling IO errors
  - A try-catch with resources
  - The "File" class. File operations
  - NIO.2

## The concept of input-output  streams 1/3

- An *I/O Stream* represents an input source or an output destination
- A stream is a sequence of data
- A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays

## The concept of input-output streams 2/3

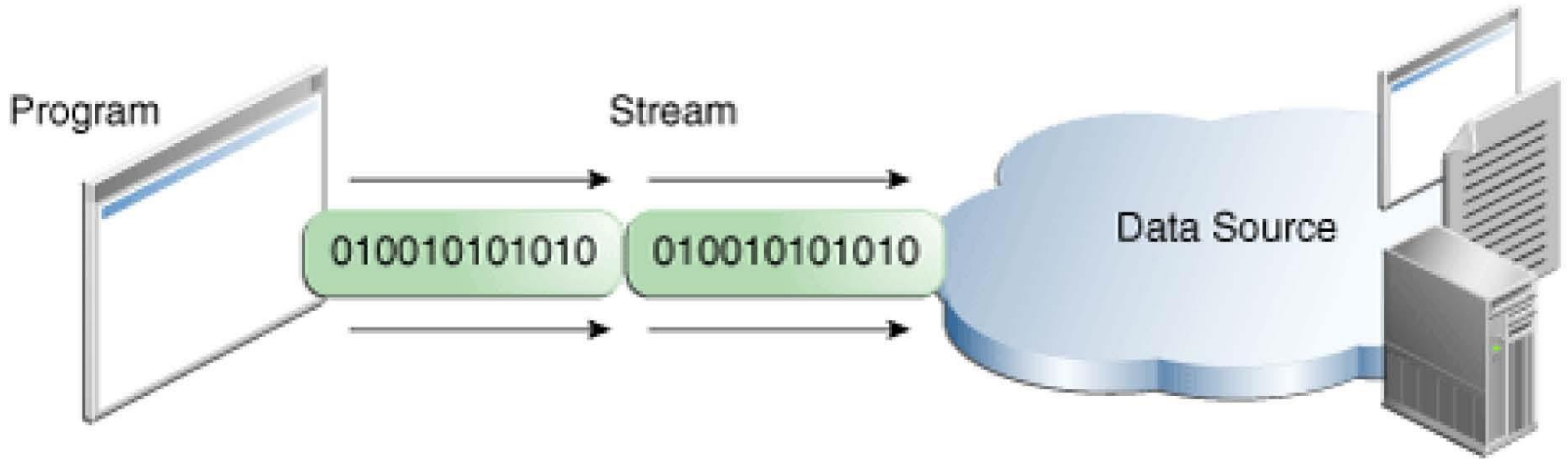
- A program uses an *input stream* to read data from a source



- Reading information into a program

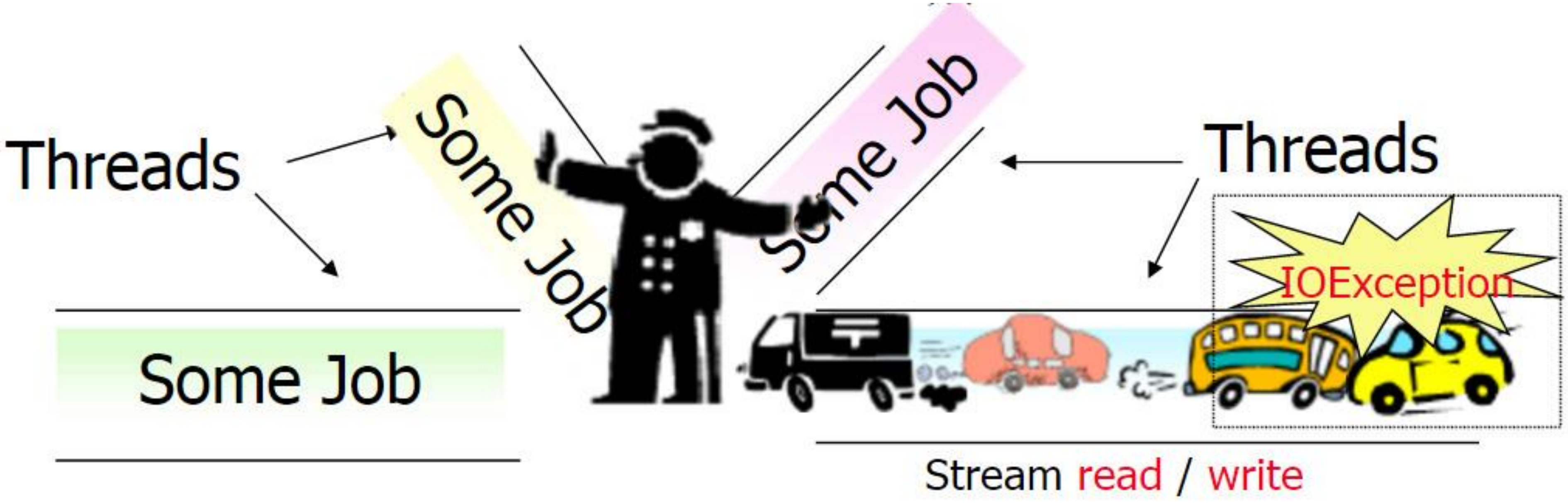
## The concept of input-output streams 3/3

- A program uses an *output stream* to write data to a destination



- Writing information from a program

# The concept of input-output stream



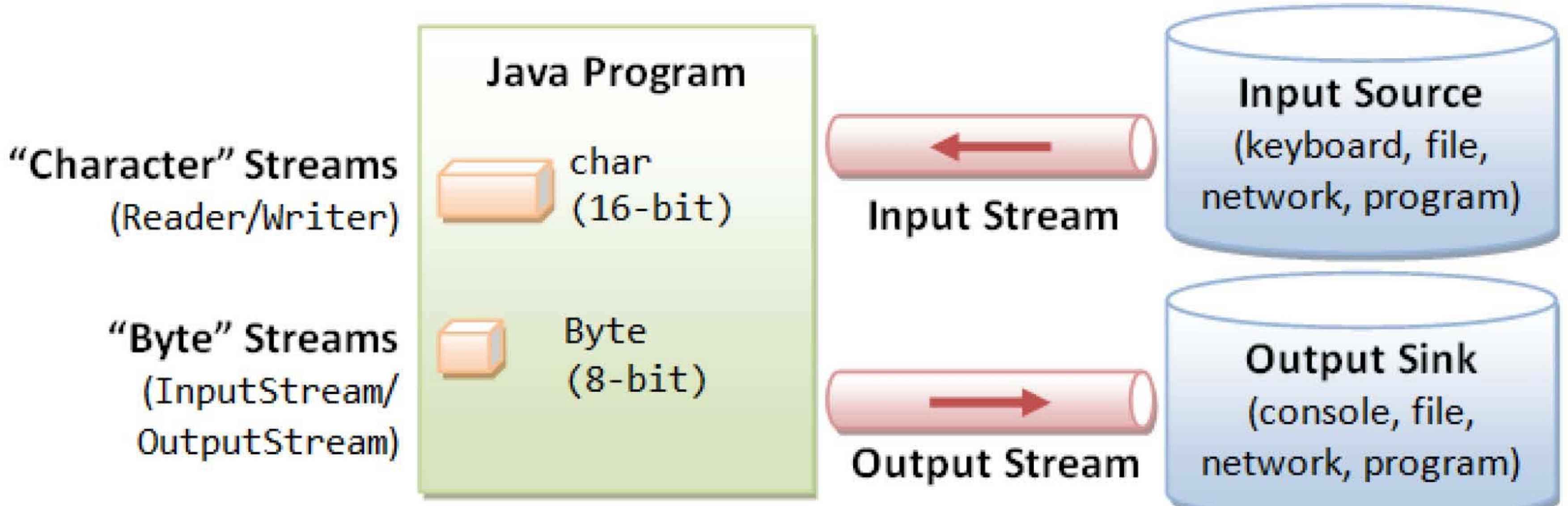
InputStream read() methods block the thread, executing these methods, until it becomes accessible portion of the read data, or the end of the InputStream is reached or an IOException is thrown.

OutputStream write() methods block the thread, executing these methods, at the time of writing data portion, or until the end of the OutputStream is reached or an IOException is thrown.

## Module contents

- Java I/O Streams
  - The concept of input-output  streams
  - Byte streams and character streams
  - The main I/O stream classes
  - The RandomAccessFile class
  - Principles of handling IO errors
  - A try-catch with resources
  - The "File" class. File operations
  - NIO.2

## Byte streams and character streams



Internal Data Formats:

- Text (char): UCS-2
- int, float, double, etc.

External Data Formats:

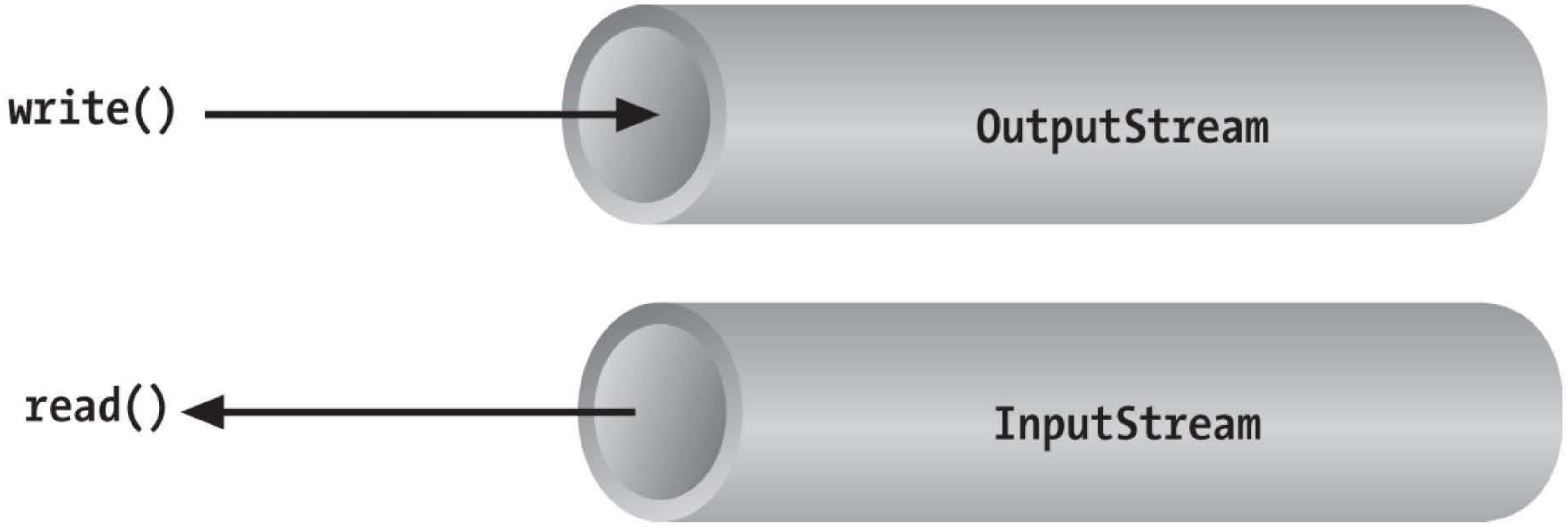
- Text in various encodings (US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
- Binary (raw bytes)

## Module contents

- Java I/O Streams
  - The concept of input-output  streams
  - Byte streams and character streams
  - The main I/O stream classes
  - The RandomAccessFile class
  - Principles of handling IO errors
  - A try-catch with resources
  - The "File" class. File operations
  - NIO.2

## The main I/O stream classes 1/16

- InputStream and OutputStream are abstract classes that define the lowest-level interface for all byte streams



# The main I/O stream classes 2/16

| Modifier and Type | Method and Description |
|---|---|
| abstract int | **read()**<br>Reads the next byte of data from the input stream. |
| int | **read(byte[]b)**<br>Reads some number of bytes from input stream and stores them into the buffer array b. |
| int | **read(byte[]b, int off, int len)**<br>Reads up to len bytes of data from the input  stream into array of bytes. |
| void | **reset()**<br>Repositions this stream to the position at the time the mark method was last called on this input stream. |
| long | **skip(long n)**<br>Skips over and discards n bytes of data from this input stream |

# The main I/O stream classes 3/16

- Methods of InputStream class

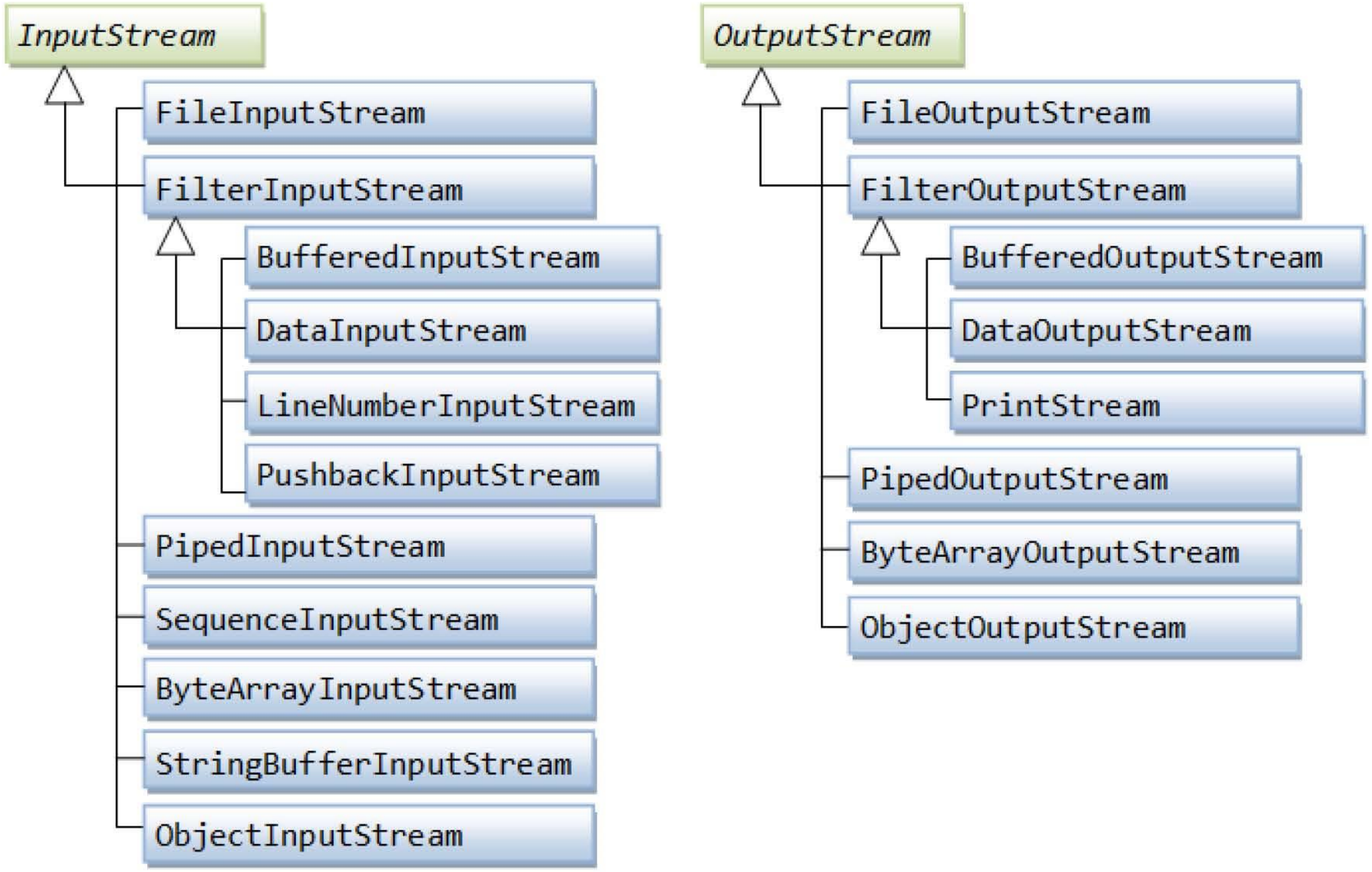
| Modifier and Type | Method and Description |
|---|---|
| int | **available()**<br>Returns an estimate of the number of bytes that can be read(or skipped over) from this input stream without blocking by next invocation of a method for this stream |
| void | **close()**<br>Closes this input stream and releases any system recourses associated with stream. |
| void | **mark(int readlimit)**<br>Marks the current position in this input stream. |
| boolean | **markSupported()**<br>Tests if this input stream supports the mark and reset methods. |

## The main I/O stream classes 4/16

- Methods of OutputStream class

| Modifier and Type | Method and Description |
|---|---|
| void | **close()**<br>Closes this output stream and releases any system recourses associated with stream. |
| void | **flush()**<br>Flushes this output stream and forces any buffered output bytes to be written out. |
| void | **write(byte[]b)**<br>Writes b.length bytes from the specified byte array to this output stream. |
| void | **write(byte[]b, int off, int len)**<br>Writes len bytes from the specified byte array starting at offset off to this output stream. |
| abstract void | **write(int b)**<br>Writes the specified byte to this output stream. |

## The main I/O stream classes 5/16



## The main I/O stream classes 6/16

```java
public class Main {
    public static void main(String[] args) {
        InputStream stdin = System.in;
        try {
            int val = System.in.read();
            System.out.println(val);
        } catch ( IOException e ) {
            //...
        }
    }
}
```

**Console output**

```
a
97
```

## The main I/O stream classes 7/16

```
public class Main {
    public static void main(String[] args) {
        OutputStream stdout = System.out;
        try {
            stdout.write(new byte[]{97,98,99});
        } catch ( IOException e ) {
            //...
        }finally {
            //...
        }
    }
}
```

**Console output**

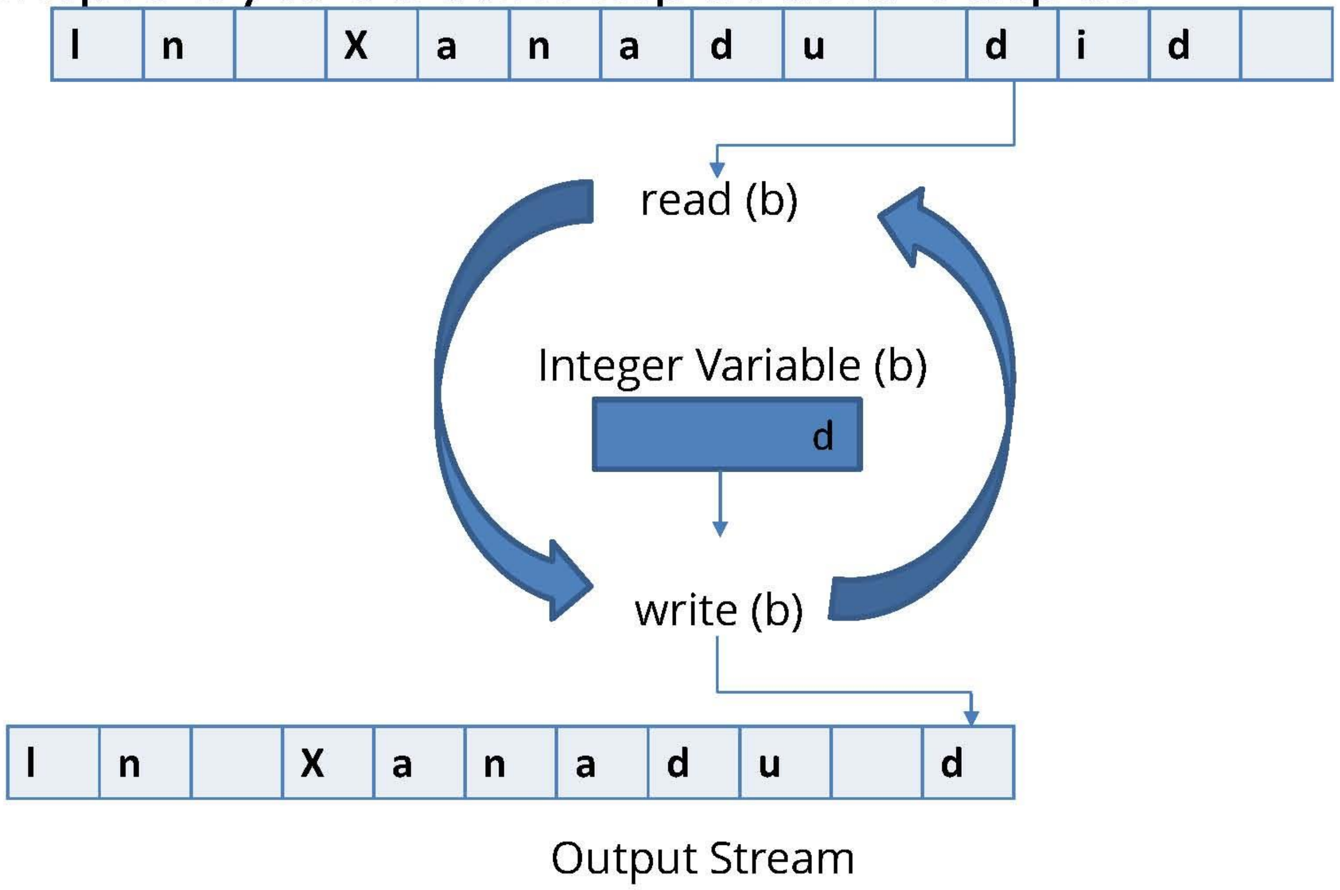
```
abc
```

## The main I/O stream classes 8/16

```
FileInputStream in = null;
FileOutputStream out = null;
try {
    in = new FileInputStream("input.txt");
    out = new FileOutputStream("output.txt");
    int c;
    while ((c = in.read()) != -1) {
        out.write(c);
    }
} finally {
    if (in != null) in.close();
    if (out != null) out.close();
}
```

## The main I/O stream classes 9/16

- Simple byte stream input and output



# abstract class Reader implements Readable, Closeable

| Modifier and Type | Method and Description |
|---|---|
| int | **read()** Reads a single character. Returns: The character read, as an integer in the range 0 to 65535 (0x00-0xffff), or -1 if the end of the stream has been reached |
| int | **read(char[] cbuf)** Reads characters into an array. |
| abstract int | **read(char[] cbuf, int off, int len)** Reads characters into a portion of an array. |
| int | **read(CharBuffer target)** Attempts to read characters into the specified character buffer. |
| boolean | **ready()** Tells whether this stream is ready to be read. |
| abstract void | **close()** Closes the stream and releases any system resources associated with it. Once the stream has been closed, further read(), ready(), mark(), reset(), or skip() invocations will throw an IOException. |

# abstract class Reader implements Readable, Closeable

| Modifier and Type | Method and Description |
|---|---|
| void | **mark(int readAheadLimit)** Marks the present position in the stream. Subsequent calls to reset() will attempt to reposition the stream to this point. |
| boolean | **markSupported()** Tells whether this stream supports the mark() operation. |
| void | **reset()** Resets the stream. If the stream has been marked, then attempt to reposition it at the mark. If the stream has not been marked, then attempt to reset it in some way appropriate to the particular stream, for example by repositioning it to its starting point. |
| long | **skip(long n)** Skips characters. This method will block until some characters are available, an I/O error occurs, or the end of the stream is reached. |

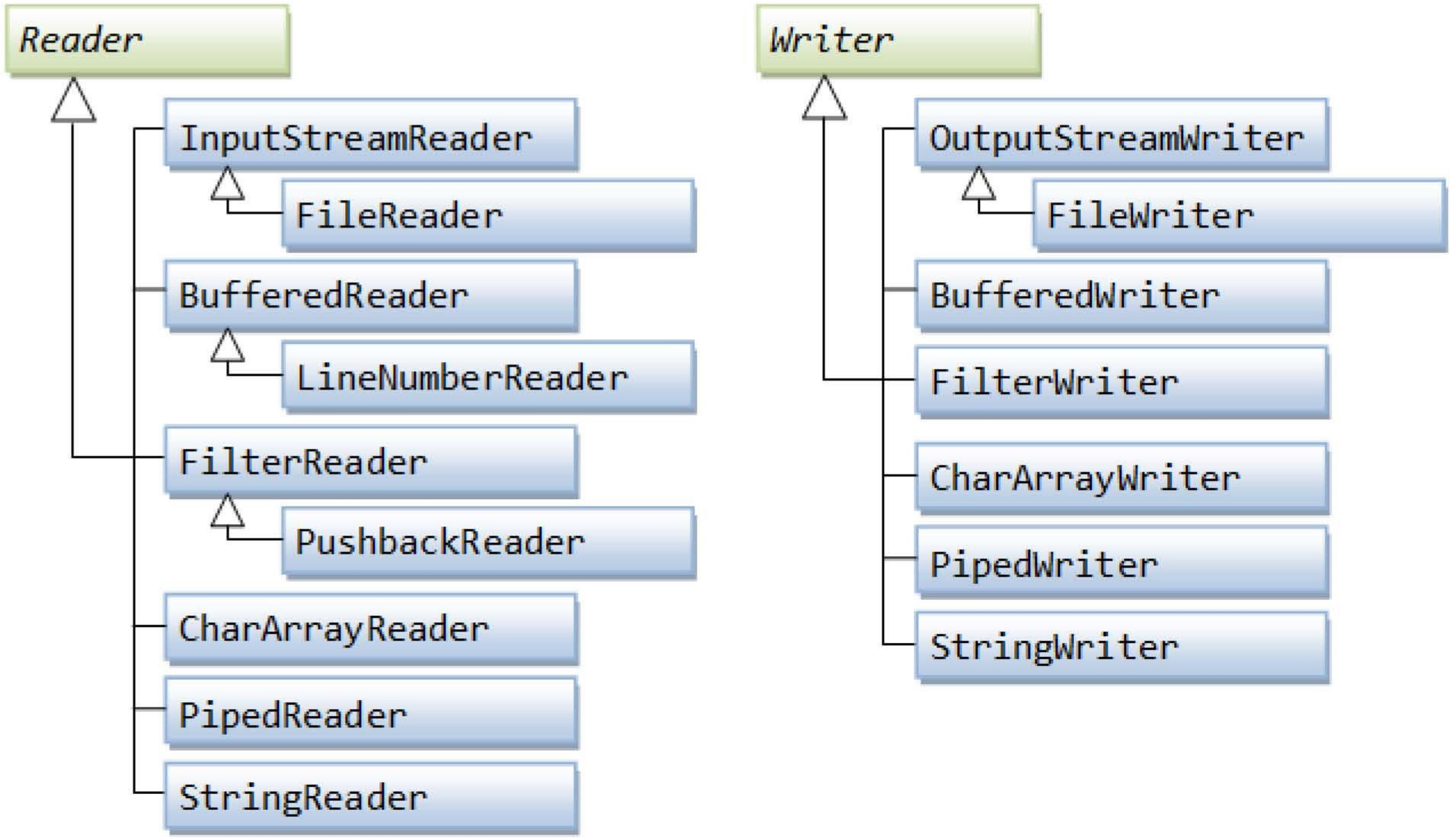
# abstract class Writer implements Appendable, Closeable, Flushable

| Modifier and Type | Method and Description |
|---|---|
| void | **write(int c)** Writes a single character. The character to be written is contained in the 16 low-order bits of the given integer value |
| void | **write(char[] cbuf)** Writes an array of characters. |
| abstract void | **write(char[] cbuf, int off, int len)** Writes a portion of an array of characters. |
| void | **write(String str)** Writes a string. |
| void | **write(String str, int off, int len)** Writes a portion of a string. |

# abstract class Writer implements Appendable, Closeable, Flushable

| Modifier and Type | Method and Description |
|---|---|
| Writer | **append(char c)** Appends the specified character to this writer. An invocation of this method of the form **out.append(c)** behaves in exactly the same way as the invocation **out.write(c)** |
| Writer | **append(CharSequence csq)** Appends the specified character sequence to this writer. |
| Writer | **append(CharSequence csq, int start, int end)** Appends a subsequence of the specified character sequence to this writer. |
| abstract void | **close()** Closes the stream, flushing it first. |
| abstract void | **flush()** Flushes the stream. If the stream has saved any characters from the various write() methods in a buffer, write them immediately to their intended destination. |

## The main I/O stream classes 10/16

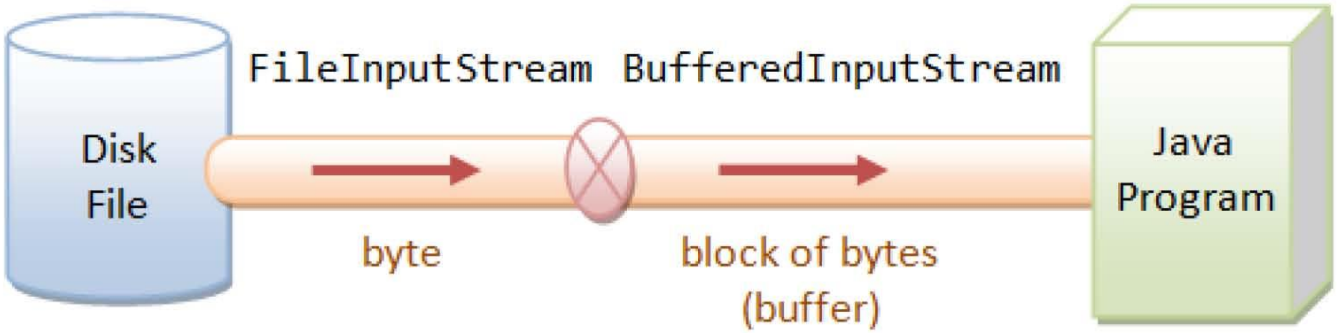


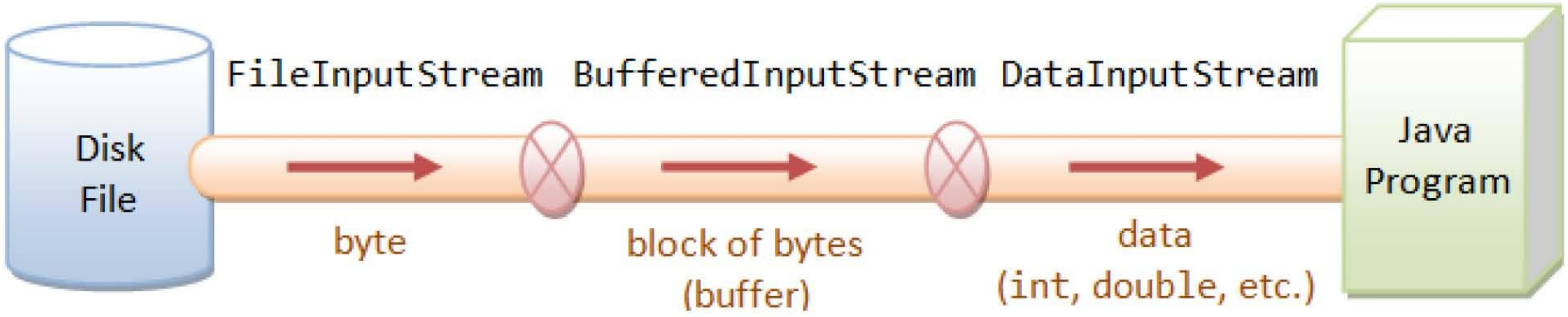
## The main I/O stream classes 11/16

```java
FileReader inputStream = null;
FileWriter outputStream = null;
try {
    inputStream = new FileReader("input.txt");
    outputStream = new FileWriter("output.txt");
    int c;
    while ((c = inputStream.read()) != -1) {
        outputStream.write(c);
    }
} finally {
    if (inputStream != null) inputStream.close();
    if (outputStream != null) outputStream.close();
}
```

## The main I/O stream classes 12/16

- Layered (or Chained) I/O Streams





## The main I/O stream classes 13/16

```
BufferedInputStream in = null;
BufferedOutputStream out = null;
try {
   in = new BufferedInputStream(new FileInputStream("input.txt"));
   out = new BufferedOutputStream(new FileOutputStream("output.txt"));
   int byteRead;
   while ((byteRead = in.read()) != -1) {
      out.write(byteRead);
   }
} catch (IOException ex) {
   //...
} finally {
   try {
      if (in != null) in.close();
      if (out != null) out.close();
   } catch (IOException ex) {
      //...
   }
}
```

## InputStreamReader

```
public class Main {
    public static void main(String[] args) {
        char[] chars = new char[12];      //char buffer
        System.out.println("Input line and press Enter:");
        try (InputStreamReader br = new                    //may be "UTF-8"
                                        InputStreamReader(System.in, "CP1251")) {
            int count = br.read(chars);
            System.out.println("Received " + count + " characters: " +
Arrays.toString(chars));
} catch (IOException ex) {
           Logger.getLogger(Main.class.getName()).log(Level.SEVERE,
null, ex);
        }
    }
}
```

# DataInput interface

| Modifier and Type | Method and Description |
|---|---|
| boolean | readBoolean() Reads one input byte and returns true if that byte is nonzero,false if that byte is zero. |
| byte | readByte() Reads and returns one input byte. |
| char | readChar() Reads two input bytes and returns a char value. |
| double | readDouble() Reads eight input bytes and returns a double value. |
| float | readFloat() Reads four input bytes and returns a float value. |
| void | readFully(byte[] b) Reads some bytes from an input stream and stores them into the buffer array b. |
| void | readFully(byte[] b, int off, int len) Reads len bytes from an input stream. |
| int | readInt() Reads four input bytes and returns an int value. |

# DataInput interface

| Modifier and Type | Method and Description |
|---|---|
| String | readLine() Reads the next line of text from the input stream. |
| long | readLong() Reads eight input bytes and returns a long value. |
| short | readShort() Reads two input bytes and returns a short value. |
| int | readUnsignedByte() Reads one input byte, zero-extends it to type int, and returns the result, which is therefore in the range 0 through 255. |
| int | readUnsignedShort() Reads two input bytes and returns an int value in the range 0through 65535. |
| String | readUTF() Reads in a string that has been encoded using a modified UTF-8 format. |
| int | skipBytes(int n) Makes an attempt to skip over n bytes of data from the input stream, discarding the skipped bytes. |

## DataOutput interface

| Modifier and Type | Method and Description |
|---|---|
| void | write(byte[] b)Writes to the output stream all the bytes in array b. |
| void | write(byte[] b, int off, int len)Writes len bytes from array b, in order, to the output stream. |
| void | write(int b)Writes to the output stream the eight low-order bits of the argument b. |
| void | writeBoolean(boolean v)Writes a boolean value to this output stream. |
| void | writeByte(int v)Writes to the output stream the eight low-order bits of the argument v. |
| void | writeBytes(String s)Writes a string to the output stream. |
| void | writeChar(int v)Writes a char value, which is comprised of two bytes, to the output stream. |

## DataOutput interface

| Modifier and Type | Method and Description |
|---|---|
| void | writeChars(String s)Writes every character in the string s, to the output stream, in order, two bytes per character. |
| void | writeDouble(double v)Writes a double value, which is comprised of eight bytes, to the output stream. |
| void | writeFloat(float v)Writes a float value, which is comprised of four bytes, to the output stream. |
| void | writeInt(int v)Writes an int value, which is comprised of four bytes, to the output stream. |
| void | writeLong(long v)Writes a long value, which is comprised of eight bytes, to the output stream. |
| void | writeShort(int v)Writes two bytes to the output stream to represent the value of the argument. |
| void | writeUTF(String s)Writes two bytes of length information to the output stream, followed by the modified UTF-8 representation of every character in the string s. |

## The main I/O stream classes 14/16

```
try {
    DataOutputStream out = new DataOutputStream(
    new BufferedOutputStream(
        new FileOutputStream("dataout.dat")));
    out.writeShort(1200);
    out.writeInt(50000);
    out.writeLong(12345678L);
    out.writeDouble(55.66);
    out.writeBoolean(true);
    out.writeUTF("Hello!!!");
    out.flush();
} catch (IOException ex) {
    //...
}
//...
```

## The main I/O stream classes 15/16

```java
try {
    DataInputStream in = new DataInputStream(
            new BufferedInputStream(
                    new FileInputStream("dataout.dat")));
    System.out.println("short:   " + in.readShort());
    System.out.println("int:     " + in.readInt());
    System.out.println("long:    " + in.readLong());
    System.out.println("double:  " + in.readDouble());
    System.out.println("boolean: " + in.readBoolean());
    System.out.println("String UTF: " + in.readUTF());
    System.out.println();
} catch (IOException ex) {
    //...
}
//...
```

## The main I/O stream classes 16/16

**Console output:**

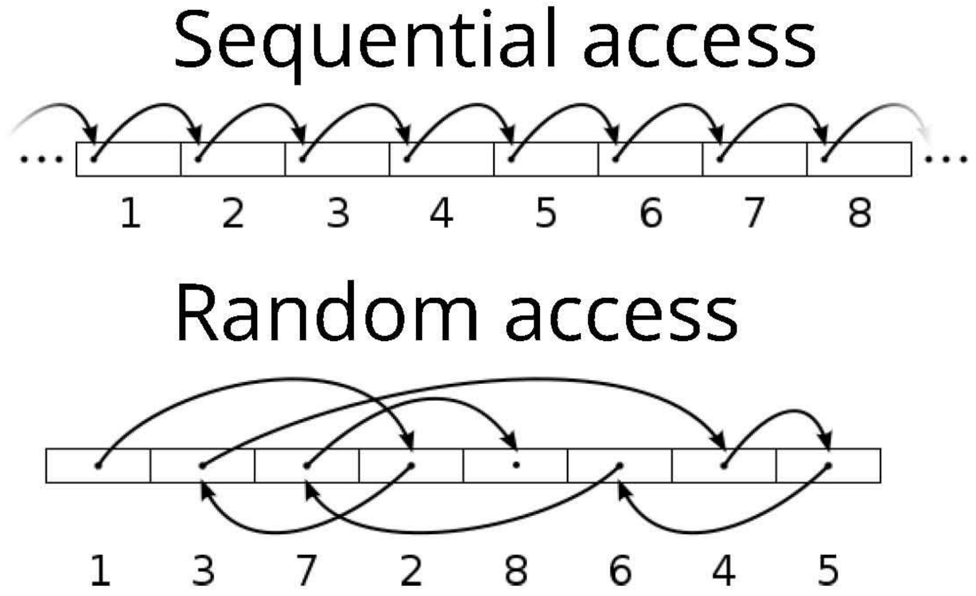
```
short:   1200
int:     50000
long:    12345678
double:  55.66
boolean: true
String UTF: Hello!!!
```

## Module contents

- Java I/O Streams
  - The concept of input-output  streams
  - Byte streams and character streams
  - The main I/O stream classes
  - The RandomAccessFile class
  - Principles of handling IO errors
  - A try-catch with resources
  - The "File" class. File operations
  - NIO.2

## The RandomAccessFile class 1/3

- RandomAccessFile. Instances of this class support both reading and writing to a random access file.
- A random access file behaves like a large array of bytes stored in the file system.



## RandomAccessFile creation

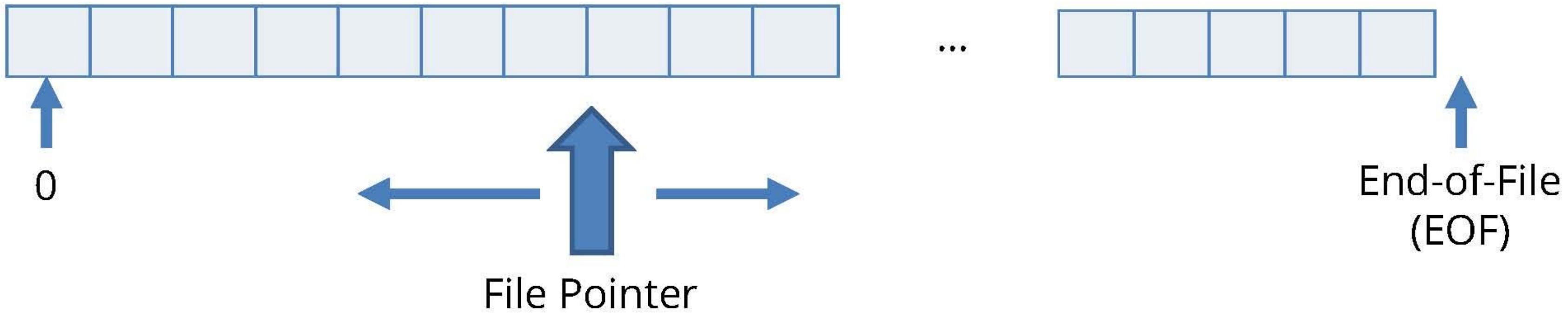
```
RandomAccessFile raf =
                new RandomAccessFile(String name, String mode)
                throws FileNotFoundException
```

### Mode values:

| Value | Meaning |
|---|---|
| "r" | Open for reading only. Invoking any of the write methods of the resulting object will cause an IOException to be thrown. |
| "rw" | Open for reading and writing. If the file does not already exist then an attempt will be made to create it. |
| "rws" | Open for reading and writing, as with "rw", and also require that every update to the file's content or metadata be written synchronously to the underlying storage device. |
| "rwd" | Open for reading and writing, as with "rw", and also require that every update to the file's content be written synchronously to the underlying storage device. |

## The RandomAccessFile class 2/3

- There is a kind of cursor, or index into the implied array, called the *file pointer*; input operations read bytes starting at the file pointer and advance the file pointer past the bytes read.



## The RandomAccessFile class 3/3

```java
RandomAccessFile raf = null;
try
{
//...
        raf = new RandomAccessFile("C:\\ra_test.txt", "rw");
        raf.write(new byte[]{0,1,2,3,4,5,6,7,8,9});
        raf.seek(5);
        raf.write(new byte[]{66,77,88});
        raf.seek(0);
        byte[] buf = new byte[10];
        int n  = raf.read(buf,0,10);
        System.out.println(Arrays.toString(buf));
        raf.close();
```

- //...

**Console output**

```
[0, 1, 2, 3, 4, 66, 77, 88, 8, 9]
```

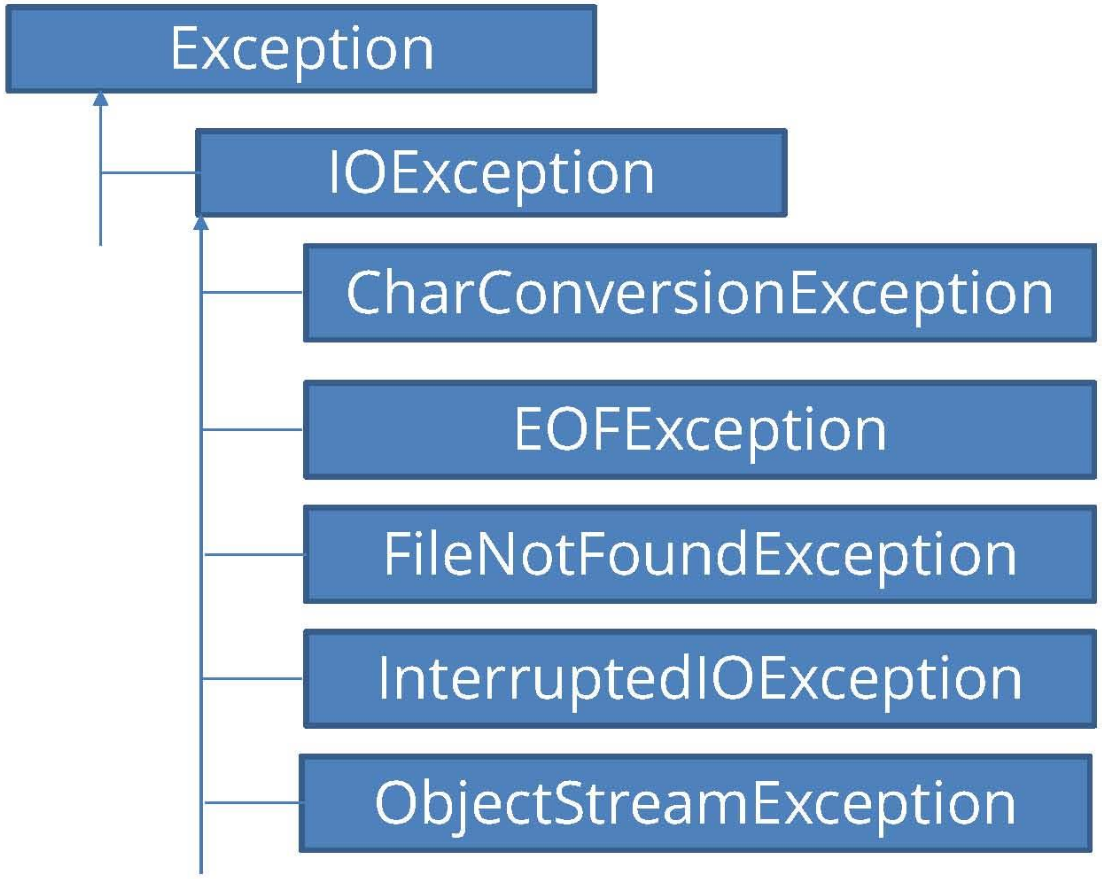
## Module contents

- Java I/O Streams
  - The concept of input-output  streams
  - Byte streams and character streams
  - The main I/O stream classes
  - The RandomAccessFile class
  - Principles of handling IO errors
  - A try-catch with resources
  - The "File" class. File operations

## Principles of handling IO errors 1/3

- **IOException** is the general class of exceptions produced by failed or interrupted I/O operations
- **IOException** signals that an I/O exception of some sort has occurred.
- Streams need to be closed properly when you are done using them. This is done by calling the close() method.

## Principles of handling IO errors 2/3



# Principles of handling IO errors 3/3

- Try-Catch-Finally, Old School Style

```
InputStream input = null;
try {
    input = new FileInputStream("file.txt");//FileNotFoundException
    int data = input.read();     //IOException
    while(data != -1){
        System.out.print((char) data);
        data = input.read();  //IOException
    }
} finally {
    if(input != null){
        input.close(); //IOException
    }
}
```

**this is a problem!!!**

```
finally {
    try {
        is.close();
    } catch (IOException ex) {
System.out.println(ex.getMessage());
    }   }
```

## Module contents

- Java I/O Streams
  - The concept of input-output  streams
  - Byte streams and character streams
  - The main I/O stream classes
  - The RandomAccessFile class
  - Principles of handling IO errors
  - A try-catch with resources
  - The "File" class. File operations

## Principles of handling IO errors 1/2

- From Java 7 on and forward Java contains a new exception handling mechanism called "try with resources".
- This exception handling mechanism is especially targeted at handling exception handling when you are using resources that need to be closed properly after use.
- interface java.lang.AutoCloseable

## Principles of handling IO errors 2/2

**try-with-resources:**

```
                                                      //FileNotFoundException
try (InputStream is = new FileInputStream("file.txt")) {
    int data = is.read();       //IOException
    while (data != -1) {
      System.out.println((char) data);
      data = is.read();     //IOException
    }
  } catch (FileNotFoundException ex) {
    System.out.println(ex.getMessage());
  } catch (IOException ex) {
    System.out.println(ex.getMessage());
  }
}
```

## Module contents

- Java I/O Streams
  - The concept of input-output  streams
  - Byte streams and character streams
  - The main I/O stream classes
  - The RandomAccessFile class
  - Principles of handling IO errors
  - A try-catch with resources
  - The "File" class. File operations

## The "File" class. File operations 1/3

- Most of the classes defined by **java.io** operate on streams, the **File** class does not.
- **File** deals directly with files and the file system. That is, the **File** class does not specify how information is retrieved from or stored in files; it describes the properties of a file itself.

## The "File" class. File operations 2/3

- The File class in the Java IO API gives you access to the underlying file system. Using the File class you can:

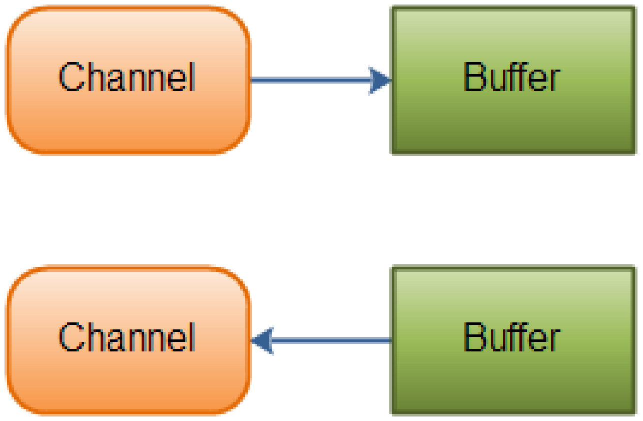
- Check if a file or directory exists.
- Create a directory if it does not exist.
- Read the length of a file.
- Rename or move a file.
- Delete a file.
- Check if path is file or directory.
- Read list of files in a directory.

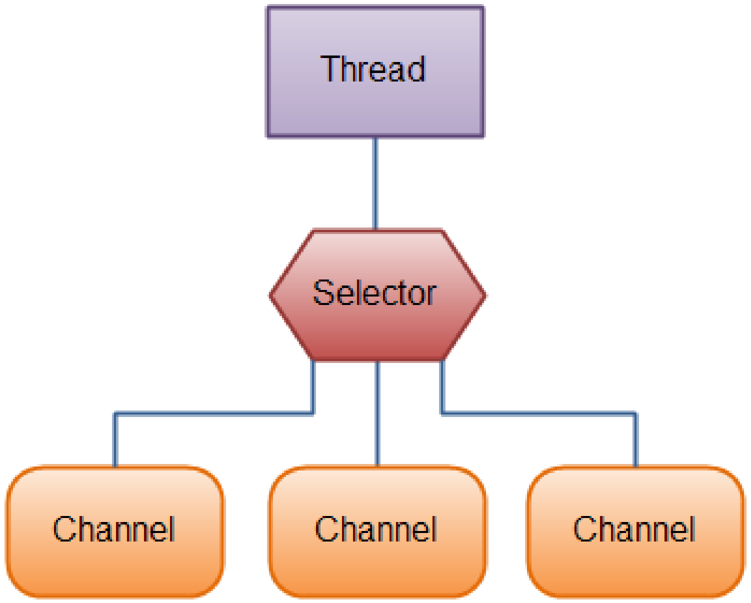
## The "File" class. File operations 3/3

```java
File file = new File("c:\\testfile.txt");
//Check if Path is File or Directory
boolean isDirectory = file.isDirectory();
//Check if File Exists
boolean fileExists = file.exists();
//File Length
long length = file.length();
//Rename or Move File
boolean sucMov =
        file.renameTo(newFile("c:\\newfile.txt"));
//Delete File
boolean successDel = file.delete();
```

# Java NIO (Non-blocking IO

| IO | NIO |
|---|---|
| Потокоориентированный | Буфер-ориентированный |
| Блокирующий (синхронный) ввод/вывод | Неблокирующий (асинхронный) ввод/вывод |
| | Селекторы |





# Java NIO

```
try (RandomAccessFile aFile = new RandomAccessFile("file.txt", "rw");) {
    FileChannel inChannel = aFile.getChannel();
    ByteBuffer buf = ByteBuffer.allocate(48);
    int bytesRead = inChannel.read(buf);
        while (bytesRead != -1) {
            System.out.println("Read " + bytesRead);
            buf.flip();
            while (buf.hasRemaining()) {
                System.out.print((char) buf.get());
            }
            buf.clear();
            bytesRead = inChannel.read(buf);
        }
    } catch (FileNotFoundException ex) {
        ...
    } catch (IOException ex) {
        ...
    }   }
```

## Java NIO - Path

- Since Java 7, the tools for working with files and directories have changed. This was due to the limitations of the **java.io.File** class. For example, it lacks a **copy()** method that allows you to copy a file from one location to another (evidently, it's a necessary function).
- In addition, the **java.io.File** class has many methods that return **boolean** values. On error, such a method returns false, rather than throwing an exception, which makes diagnosing errors and determining their causes very difficult.
- Instead of a single **java.io.File** class, the **java.nio.file.Path** interface and the **java.nio.file.Paths** and **java.nio.file.Files** classes appeared.

## Java NIO - Paths

- **java.nio.file.Paths** is a simple class that was created solely in order to get an object of type Path from the passed string or URI.

```
public final class Paths {
    private Paths() { }
    public static Path get(String first, String... more) {
        return Path.of(first, more);
    }
    public static Path get(URI uri) {
        return Path.of(uri);
    }
}
```

since Java 7

# Java NIO - Path

- The **java.nio.file.Path** interface supports two types of operations: ***syntactic operations*** (almost any operation that involves manipulating paths without accessing the file system) and ***operations over files*** referenced by paths.
- The **java.nio.file.Path** interface is an upgraded version of the well-known **java.io.File** class, but the File class has kept a few specific operations, so it is not deprecated and cannot be considered obsolete.
- The **Path** object is a programmatic representation of a path in the file system

```
Path basePath = Paths.get(“C:/Users/kgp/data");
Path fullPath = basePath.resolve("index.html");
System.out.println(“Full path: “ + fullPath);
                                    //Full path: C:\Users\kgp\data\index.html
```

since Java 7

## Java NIO - Path

```
Path path = Paths.get("C:/Windows/System32/cmd.exe");
System.out.println("Path detalization: " + path);
                              //Path detalization: C:\Windows\System32\cmd.exe
System.out.println("toString(): " + path.toString());
                                                 //C:\Windows\System32\cmd.exe
System.out.println("getFileName(): " + path.getFileName());       //cmd.exe
System.out.println("getName(int index): " + path.getName(0));     //Windows
System.out.println("getNameCount(): " + path.getNameCount());  //3
System.out.println("subpath(0,2): " + path.subpath(0, 2));
                                                          //Windows\System32
System.out.println("getParent(): " + path.getParent());
                                                      //C:\Windows\System32
System.out.println("getRoot(): " + path.getRoot());   // C:\
System.out.println(path.endsWith("System32/cmd.exe"));    //true
System.out.println(path.startsWith("C:/Windows"));         //true
System.out.println(path.isAbsolute());                     //true
```

# Java NIO - Path

```
Path dir1 = Paths.get("/home/./joe/foo");
Path dir2 = Paths.get("/home/sally/../joe/foo");
System.out.println(dir1.normalize());      //\home\joe\foo
System.out.println(dir2.normalize());       //\home\joe\fo

String file = "manifest.mf";    //in the root of the application
Path p6 = Paths.get(file);
System.out.println("Absolute path to " + file + " is: " + p6.toAbsolutePath());
//check the real file path
try {
   System.out.println("Real path to " + file + " is: " + p6.toRealPath());
} catch (IOException ex) {
   System.out.println("There is not " + file + " in the root directory of this
application");
}
```

# Java NIO - Path

```
/* Combining two paths */
Path p7 = Paths.get("/var");
System.out.format("%s%n", p7.resolve("log"));      // \var\log

/* Creating a path between two paths */
Path p8 = Paths.get("C:\\Users");
Path p9 = Paths.get("C:\\Users\\Desktop\\testFile.txt");
System.out.println(p8.relativize(p9));   //Desktop\testFile.txt
```

# Java NIO - Files

- NIO.2 comes with a set of brand new methods to accomplish the most common tasks for managing files and directories, such as create, read, write, move, delete, and so on, most of which are found in the **java.nio.file.Files** class.

```
Path testFile1 =
Files.createFile(Paths.get("C:\\Users\\Username\\Desktop\\testFile111.txt"));
 (public static Path createDirectory(Path dir, FileAttribute<?>... attrs) throws IOException
)
Path basePath = Paths.get("src/nio/filespath/data");
Path filePath = basePath.resolve("logging.properties");

/*Check if the file exists*/
LinkOption[] linkOptions = new LinkOption[]{LinkOption.NOFOLLOW_LINKS};
boolean pathExists = Files.exists(filePath, linkOptions});
System.out.println("File " + filePath + (pathExists == true ? " is exist“
                                                    : " doesn't exist"));
                  //File src\nio\filespath\data\logging.properties is exist
```

# Java NIO - Files

```
…
/*Create subdirectory*/
Path subDirPath = basePath.resolve("subdir");
try {
    Path newDir = null;
    if (Files.exists(basePath, linkOptions)) {
        newDir = Files.createDirectory(subDirPath);
        System.out.println("Subdirectory " + newDir + " is created");
    } else {
        System.out.println("Parent directory " + basePath+ " doesn't exist");
    }
} catch (FileAlreadyExistsException e) {
    // the directory already exists
} catch (IOException e) {
    //something else went wrong
    e.printStackTrace();
}
```

```
//Subdirectory src\nio\filespath\data\subdir is created
```

# Java NIO - Files

```
...
/*File copy*/
Path sourcePath = filePath;
Path destinationPath = basePath.resolve("logging-copy.properties");
try {
    if (Files.exists(sourcePath, linkOptions)
            && Files.exists(destinationPath.getParent(), linkOptions)) {
        Files.copy(sourcePath, destinationPath);
        System.out.println("File " + sourcePath + " is copied to "
                                                            + destinationPath);
    } else {
        System.out.println("Source file path " + sourcePath
                + " or/and destination parent directory path "
                + destinationPath.getParent() + " is wrong");
    }                                       //File src\nio\filespath\data\logging.properties
} catch (IOException e) {                   //is copied to src\nio\filespath\data\logging-
    e.printStackTrace();                    //copy.properties
}
```

# Java NIO - Files

```
…
/*File copy with replace*/
sourcePath = filePath;
destinationPath = basePath.resolve("logging-copy.properties");
try {
    if (Files.exists(sourcePath, linkOptions)
            && Files.exists(destinationPath.getParent(), linkOptions)) {
        Files.copy(sourcePath, destinationPath,
                StandardCopyOption.REPLACE_EXISTING);
        System.out.println("File " + sourcePath + " is copied with replace to "
                                                        + destinationPath);
    } else {
        System.out.println("Source file path " + sourcePath
                + " or/and destination parent directory path "
                + destinationPath.getParent() + " is wrong");
    }
} catch (IOException e) {      //File src\nio\filespath\data\logging.properties
    e.printStackTrace();       //is copied with replace to
}                              //src\nio\filespath\data\logging-copy.properties
```

# Java NIO - Files

```
…
/*File move*/
sourcePath = basePath.resolve("logging-copy.properties");
destinationPath = basePath.resolve("subdir/logging-moved.properties");
try {
    if (Files.exists(sourcePath, linkOptions)
            && Files.exists(destinationPath.getParent(), linkOptions)) {
        Files.move(sourcePath, destinationPath,
                StandardCopyOption.REPLACE_EXISTING);
        System.out.println("File " + sourcePath + " is moved to "
                                                     + destinationPath);
    } else {
        System.out.println("Source file path " + sourcePath
                + " or/and destination parent directory path "
                + destinationPath.getParent() + " is wrong");
    }
} catch (IOException e) {   //File src\nio\filespath\data\logging.properties
    e.printStackTrace();    // is moved to src\nio\filespath\data\subdir\
}                           l//ogging-moved.properties
```

# Java NIO - Files

```
…
/*File/directory delete – directory must be empty!!!*/
Path path = basePath.resolve("subdir/logging-moved.properties");
Path path1 = subDirPath;
try {
    if (Files.exists(path, linkOptions)) {
        Files.delete(path);
        Files.delete(path1);
        System.out.println("File " + path + " and directory " + path1
                                                  + " are deleted");
    } else {
        System.out.println("File path " + path + " is wrong");
    }
} catch (IOException e) {           // File src\nio\filespath\data\subdir\logging-
    //deleting file failed          //moved.properties and directory
    e.printStackTrace();            //src\nio\filespath\data\subdir are deleted
}
```

# Java NIO - Files

```
List<String> lines =
Files.readAllLines(Paths.get("C:\\Users\\Username\\Desktop\\pushkin.txt"),
UTF_8);
for (String s: lines) {
    System.out.println(s);
}

Stream<String> stream =
Files.lines(Paths.get("C:\\Users\\Username\\Desktop\\pushkin.txt"));
List<String> result  = stream
        .filter(line -> line.startsWith("Как"))
        .map(String::toUpperCase)
        .collect(Collectors.toList());
result.forEach(System.out::println);
```