NIO 2

Java NIO

- NIO stands for New IO.

- Java NIO consist of the following core components:
  - Channels
  - Buffers
  - Selectors
  - Pipe
  - FileLock

Difference between NIO and IO

NIO

- Buffer oriented
- Can move back and forth
- Non-blocking IO operations

IO

- Stream oriented
- Can move only forth
- Blocking IO operations

What is Channel?

- Channels are used for I/O transfers.
- Channel is a like a tube that transports data between a buffer and an entity at other end.
- A channel reads data from an entity and places it in buffer blocks for consumption.
- Similarly, Data can be written to buffer blocks and that data will be transported by the channel to the other end.

Channel Characteristics

- Unlike streams, channels are two-way. A channel can both read and write.
- Channel reads data into a buffer and writes data from a buffer.
- Channels can do asynchronous read and write operations.
- Channels can be on blocking or non-blocking modes.
- Non-blocking channel does not put the invoking thread in sleep mode.
- Stream-oriented channels like sockets only can be placed in non-blocking mode.
- Data can be transferred from Channel to Channel if any one of them is a FileChannel.

Java NIO Channel Classes

Two major type of Channel classes are provided by Java NIO :-

- **FileChannel**
  - These are based File read/write channels that cannot be placed on nonblocking mode.

- **SocketChannel**
  - There are three socket channel types namely, SocketChannel, ServerSocketChannel and DatagramChannel.

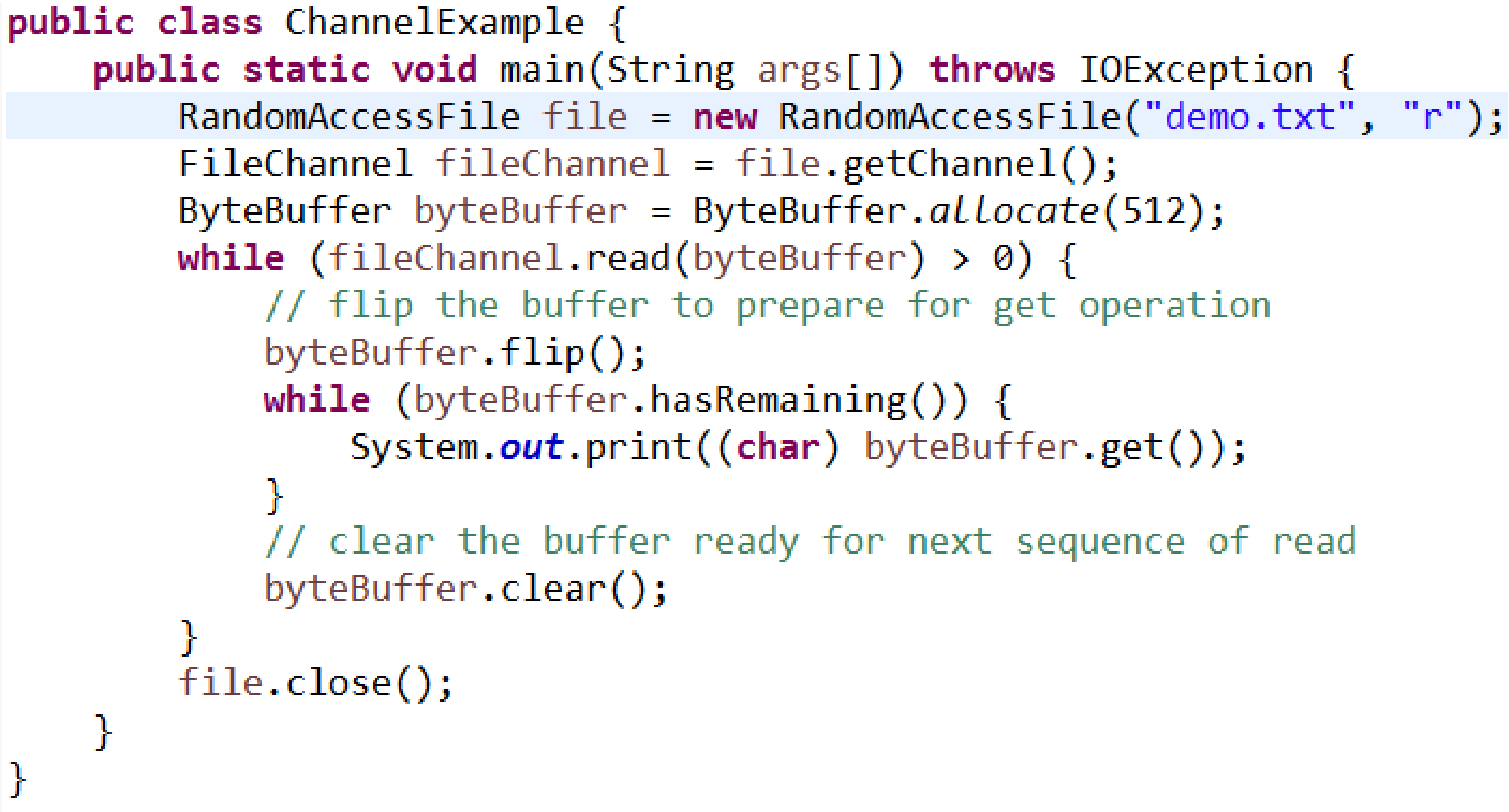
Random Access File

- Random access files permit nonsequential, or random, access to a file's contents.

- To access a file randomly, open the file, seek a particular location, and read from or write to that file.

- Java NIO 2 introduces a new interface java.nio.SeekableByteChannel which is an subinterface of OlderByteChannel interface

Example : Channel



```
public class ChannelExample {
    public static void main(String args[]) throws IOException {
        RandomAccessFile file = new RandomAccessFile("demo.txt", "r");
        FileChannel fileChannel = file.getChannel();
        ByteBuffer byteBuffer = ByteBuffer.allocate(512);
        while (fileChannel.read(byteBuffer) > 0) {
            // flip the buffer to prepare for get operation
            byteBuffer.flip();
            while (byteBuffer.hasRemaining()) {
                System.out.print((char) byteBuffer.get());
            }
            // clear the buffer ready for next sequence of read
            byteBuffer.clear();
        }
        file.close();
    }
}
```

Path

- The Java Path interface is part of the Java NIO 2 update.

- The Java Path interface was added to Java NIO in Java 7. The Path interface is located in the java.nio.file package.

- A Java Path instance represents a path in the file system. A path can point to either a file or a directory.

- A path can be absolute or relative.
  - An absolute path contains the full path from the root of the file system down to the file or directory it points to.
  - A relative path contains the path to the file or directory relative to some other path.

Path : Methods

Method Name	Description
getFileName()	Returns the file name or the last element of the sequence of name elements.
getName(int index)	Returns the path element corresponding to the specified index. The 0th element is the path element closest to the root.
getNameCount()	Returns the number of elements in the path.
subpath(int startIndex, int endIndex)	Returns the subsequence of the Path (not including a root element) as specified by the beginning and ending indexes.
getParent()	Returns the path of the parent directory.
getRoot()	Returns the root of the path.
toUri()	Returns a URI to represent this path.
normalize()	Returns a path that is this path with redundant name elements eliminated.
resolve(Path other)	Resolve the given path against this path.
relativize(String other)	Converts a given path string to a Path and resolves it against this Path in exactly the manner specified by the resolve method.

Example : Path Operations

- Path instance can be created by using a static method in the Paths class (java.nio.file.Paths) named Paths.get().

```
Path path1 = Paths.get("c:\\data\\myfile.txt");
```

```
Path path2= Paths.get("d:\\data", "projects");
```

```
Path path3 =
Paths.get(URI.create("file:///manipal/demo/MyFile.java"));
```

Example : Path Operations

```
Path path = Paths.get("c:\\data\\myfile.txt");
String filename = path.getFileName();
String subPath = path.subPath(0,2);
String parent = path.getParent();
String root = path.getRoot();
Uri uri  = path.toUri();
```

Joining two Paths

- To join two paths in NIO, resolve method is used. If an absolute path is passed as parameter to resolve method, then the same is returned.

- If partial path (which is a path that does not include a root element) is passed, it is appended to the original path.

```
Path path1 = Paths.get("C:\\Users\\Java\\examples");
System.out.println(path1.resolve("Test.java"));
// Output is C:\Users\Java\examples\Test.java
```

Bridge Paths

- To construct a Path from one location to another relativize method is used. This can be used to navigate between two paths.

```
Path path1 = Paths.get("C:\\Users");
Path path2 = Paths.get("C:\\Users\\Java\\examples");
// outcome is Java\examples
Path path1_to_path2 = path1.relativize(path2);
// outcome is ..\..
Path path2_to_path1 = path2.relativize(path1);
```

Metadata (File Attributes)

- The definition of *metadata* is "data about other data."
- A file system's metadata is typically referred to as its ***file attributes***.
- With a file system, the data is contained in its files and directories, and the metadata tracks information about each of these objects.
  - File type (a regular file, a directory, or a link)
  - Size of file,
  - creation date,
  - last modified date,
  - file owner,
  - group owner, and
  - access permissions

Files Class

- The Files class includes methods that can be used to obtain a single attribute of a file, or to set an attribute.

Methods	Description
size(Path)	Returns the size of the specified file in bytes.
isDirectory(Path, LinkOption)	Returns true if the specified Path locates a file that is a directory.
isRegularFile(Path, LinkOption...)	Returns true if the specified Path locates a file that is a regular file.
isSymbolicLink(Path)	Returns true if the specified Path locates a file that is a symbolic link.
isHidden(Path)	Returns true if the specified Path locates a file that is considered hidden by the file system.

getLastModifiedTime(Path, LinkOption...)
setLastModifiedTime(Path, FileTime)
	Returns or sets the specified file's last modified time.

getAttribute(Path, String, LinkOption...)
setAttribute(Path, String, Object, LinkOption...)
	Returns or sets the value of a file attribute.

Views

- Related file attributes are grouped together into views. A *view* maps to a particular file system implementation, such as POSIX or DOS, or to a common functionality, such as file ownership.

Views
BasicFileAttributeView
DosFileAttributeView
PosixFileAttributeView –
FileOwnerAttributeView
AclFileAttributeView
UserDefinedFileAttributeView

Symbolic Links

- A *symbolic link* is a special file that serves as a reference to another file.
- Symbolic links are transparent to applications, and operations on symbolic links are automatically redirected to the target of the link.
- The file or directory being pointed to is called the *target* of the link.
- Exceptions are when a symbolic link is deleted, or renamed in which case the link itself is deleted, or renamed and not the target of the link.
- A symbolic link is also referred to as a *symlink* or a *soft link*.

Hard Link

- In hard link, only an entry into directory structure is created for the file, but it points to the inode location of the original file.

- Some file systems also support hard links. *Hard links* are more restrictive than symbolic links.

Creating symbolic and hard link

- **Files class contains methods for creating symbolic and hard links**

```
//create symbolic link
Files.createSymbolicLink(Path link, Path target)
//create hard link
Files.createLink(Path link, Path existing)
//to check for symbolic link
Files.isSymbolicLink(Path path)
//Reads the target of a symbolic link
Files.readSymbolicLink(Path link)
```

Walking a File Tree

- FileVisitor is an interface and we need to implement it to walk a file tree. FileVisitor has got the following four methods,

Method	Description
preVisitDirectory	called before contents of a directory is visited.
postVisitDirectory	called after contents of a directory is visited.
visitFile	called on the file that is being visited.
visitFileFailed	called when there is a failure on visiting a file.

Walking a File Tree : SimpleFileVisitor Class

- SimpleFileVisitor is a class with default implementation to visit all files and on error it will re-throw errors.

- Instead of implementing FileVisitor, SimpleFileVisitor can be extended and override only required methods.

File Tree Walk Flow Control

- File tree navigation can be controlled.
- Every method returns FileVisitResult and value can be set to control the further action on the tree navigation using it. FileVisitResult has the following values

  - CONTINUE – File tree walking should continue.
  - TERMINATE – Abort the file tree walking.
  - SKIP_SUBTREE – The directory and its subdirectories skipped when this value is returned by preVisitDirectory.
  - SKIP_SIBLINGS – When returned from preVisitDirectory, it is same as previous option and postVisitDirectory method also not called. If returned from postVisitDirectory, then no other directories in the same level are visited.