Reading and Writing Binary Files

Reading a File by Using Stream I/O

To open a file for reading, you can use the `newInputStream(Path, OpenOption...)` method. This method returns an unbuffered input stream for reading bytes from the file.

```
Path file = ...;
try (InputStream in = Files.newInputStream(file);
     BufferedReader reader =
       new BufferedReader(new InputStreamReader(in))) {
     String line = null;
     while ((line = reader.readLine()) != null) {
         System.out.println(line);
     }
} catch (IOException x) {
     System.err.println(x);
}
```

Creating and Writing a File by Using Stream I/O

You can create a file, append to a file, or write to a file by using the `newOutputStream(Path, OpenOption...)` method. This method opens or creates a file for writing bytes and returns an unbuffered output stream.

The method takes an optional `OpenOption` parameter. If no open options are specified, and the file does not exist, a new file is created. If the file exists, it is truncated. This option is equivalent to invoking the method with the `CREATE` and `TRUNCATE_EXISTING` options.

The following example opens a log file. If the file does not exist, it is created. If the file exists, it is opened for appending.

```
import static java.nio.file.StandardOpenOption.*;
import java.nio.file.*;
import java.io.*;

public class LogFileTest {

  public static void main(String[] args) {

    // Convert the string to a
    // byte array.
    String s = "Hello World! ";
    byte data[] = s.getBytes();
    Path p = Paths.get("./logfile.txt");

    try (OutputStream out = new BufferedOutputStream(
      Files.newOutputStream(p, CREATE, APPEND))) {
      out.write(data, 0, data.length);
    } catch (IOException x) {
      System.err.println(x);
    }
  }
}
```

Reading and Writing Files by Using Channel I/O

While stream I/O reads a character at a time, channel I/O reads a buffer at a time. The `ByteChannel` interface provides basic read and write functionality. A `SeekableByteChannel` is a `ByteChannel` that has the capability to maintain a position in the channel and to change that position. A `SeekableByteChannel` also supports truncating the file associated with the channel and querying the file for its size.

The capability to move to different points in the file and then read from or write to that location makes random access of a file possible. See the section Random Access Files for more information.

There are two methods for reading and writing channel I/O.

- `newByteChannel(Path, OpenOption...)`
- `newByteChannel(Path, Set<? extends OpenOption>, FileAttribute<?>...)`

> *Note: The `newByteChannel()` methods return an instance of a `SeekableByteChannel`. With a default file system, you can cast this seekable byte channel to a `FileChannel` providing access to more advanced features such mapping a region of the file directly into memory for faster access, locking a region of the file so other processes cannot access it, or reading and writing bytes from an absolute position without affecting the channel's current position.*

Both `newByteChannel()` methods enable you to specify a list of `OpenOption` options. The same open options used by the `newOutputStream()` methods are supported, in addition to one more option: `READ` is required because the `SeekableByteChannel` supports both reading and writing.

Specifying `READ` opens the channel for reading. Specifying `WRITE` or `APPEND` opens the channel for writing. If none of these options are specified, then the channel is opened for reading.

The following code snippet reads a file and prints it to standard output:

```
public static void readFile(Path path) throws IOException {

    // Files.newByteChannel() defaults to StandardOpenOption.READ
    try (SeekableByteChannel sbc = Files.newByteChannel(path)) {
        final int BUFFER_CAPACITY = 10;
        ByteBuffer buf = ByteBuffer.allocate(BUFFER_CAPACITY);

        // Read the bytes with the proper encoding for this platform. If
        // you skip this step, you might see foreign or illegible
        // characters.
        String encoding = System.getProperty("file.encoding");
        while (sbc.read(buf) > 0) {
            buf.flip();
            System.out.print(Charset.forName(encoding).decode(buf));
```

```
                buf.clear();
            }
        }
    }
```

The following example, written for UNIX and other POSIX file systems, creates a log file with a specific set of file permissions. This code creates a log file or appends to the log file if it already exists. The log file is created with read/write permissions for owner and read only permissions for group.

```
import static java.nio.file.StandardOpenOption.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;
import java.nio.file.attribute.*;
import java.io.*;
import java.util.*;

public class LogFilePermissionsTest {

  public static void main(String[] args) {

    // Create the set of options for appending to the file.
    Set<OpenOption> options = new HashSet<OpenOption>();
    options.add(APPEND);
    options.add(CREATE);

    // Create the custom permissions attribute.
    Set<PosixFilePermission> perms =
      PosixFilePermissions.fromString("rw-r-----");
    FileAttribute<Set<PosixFilePermission>> attr =
      PosixFilePermissions.asFileAttribute(perms);

    // Convert the string to a ByteBuffer.
    String s = "Hello World! ";
    byte data[] = s.getBytes();
    ByteBuffer bb = ByteBuffer.wrap(data);

    Path file = Paths.get("./permissions.log");

    try (SeekableByteChannel sbc =
      Files.newByteChannel(file, options, attr)) {
      sbc.write(bb);
    } catch (IOException x) {
```

```
        System.out.println("Exception thrown: " + x);
      }
    }
}
```

Last update: *January 25, 2023*

[Reading and Writing Text Files](#) ← **Reading and Writing Binary Files** → [Decorating IO Streams](#)

Home > Tutorials > The Java I/O API > File Operations Basics > Reading and Writing Binary Files