FIO12-J. Provide methods to read and write little-endian data

In Java, data is stored in big-endian format (also called network order). That is, all data is represented sequentially starting from the most significant bit to the least significant. JDK versions prior to JDK 1.4 required definition of custom methods that manage reversing byte order to maintain compatibility with little-endian systems. Correct handling of byte order–related issues is critical when exchanging data in a networked environment that includes both big-endian and little-endian machines or when working with other languages using Java Native Interface (JNI). Failure to handle byte-ordering issues can cause unexpected program behavior.

Noncompliant Code Example

The read methods (`readByte()`, `readShort()`, `readInt()`, `readLong()`, `readFloat()`, and `readDouble()`) and the corresponding write methods defined by class `java.io.DataInputStream` and class `java.io.DataOutputStream` operate only on big-endian data. Use of these methods while interoperating with traditional languages, such as C and C++, is insecure because such languages lack any guarantees about endianness. This noncompliant code example shows such a discrepancy:

```
try {
  DataInputStream dis = null;
  try {
    dis = new DataInputStream(new FileInputStream("data"));
    // Little-endian data might be read as big-endian
    int serialNumber = dis.readInt();
  } catch (IOException x) {
    // Handle error
  } finally {
    if (dis != null) {
      try {
       dis.close();
      } catch (IOException e) {
      // Handle error
      }
    }
  }
}
```

Compliant Solution (`ByteBuffer`)

This compliant solution uses methods provided by class `ByteBuffer` [API 2014] to correctly extract an `int` from the original input value. It wraps the input byte array with a `ByteBuffer`, sets the byte order to little-endian, and extracts the `int`. The result is stored in the integer `serialNumber`. Class `ByteBuffer` provides analogous get and put methods for other numeric types.

```
try {
  DataInputStream dis = null;
  try {
    dis = new DataInputStream( new FileInputStream("data"));
    byte[] buffer = new byte[4];
    int bytesRead = dis.read(buffer);  // Bytes are read into buffer
    if (bytesRead != 4) {
      throw new IOException("Unexpected End of Stream");
    }
    int serialNumber =
        ByteBuffer.wrap(buffer).order(ByteOrder.LITTLE_ENDIAN).getInt();
  } finally {
    if (dis != null) {
  try {
    dis.close();
  } catch (IOException x) {
    // Handle error
  }
    }
  }
} catch (IOException x) {
  // Handle error
}
```

Compliant Solution (Define Special-Purpose Methods)

An alternative compliant solution is to define read and write methods that support the necessary byte-swapping while reading from or writing to the file. In this example, the `readLittleEndianInteger()` method reads four bytes into a byte buffer and then pieces together the integer in the correct order. The `writeLittleEndianInteger()` method obtains bytes by repeatedly casting the integer so that the least significant byte is extracted on successive right shifts. `Long` values can be handled by defining a byte buffer of size 8.

```
// Read method
public static int readLittleEndianInteger(InputStream ips)
                                          throws IOException {
  byte[] buffer = new byte[4];
  int check = ips.read(buffer);

  if (check != 4) {
    throw new IOException("Unexpected End of Stream");
  }

  int result = (buffer[3] << 24) | (buffer[2] << 16) |
               (buffer[1] << 8)  | buffer[0];
  return result;
}

// Write method
public static void writeLittleEndianInteger(int i, OutputStream ops)
  throws IOException {
  byte[] buffer = new byte[4];
  buffer[0] = (byte) i;
  buffer[1] = (byte) (i >> 8);
  buffer[2] = (byte) (i >> 16);
  buffer[3] = (byte) (i >> 24);
  ops.write(buffer);
}
```

Compliant Solution (`reverseBytes()`)

When programming for JDK 1.5 and later, use the `reverseBytes()` method defined in the classes `Character`, `Short`, `Integer`, and `Long` to reverse the order of the integral value's bytes. Note that classes `Float` and `Double` lack such a method.

```
public static int reverse(int i) {
  return Integer.reverseBytes(i);
}
```

Risk Assessment

Reading and writing data without considering endianness can lead to misinterpretations of both the magnitude and sign of the data.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO12-J	Low	Unlikely	Low	**P3**	**L3**

Automated Detection

Automated detection is infeasible in the general case.

Related Guidelines

MITRE CWE	CWE-198, Use of Incorrect Byte Ordering

Bibliography

[API 2014]	Class `ByteBuffer`
	  Method `wrap()`
	  Method `order()`
	Class `Integer`
	  Method `reverseBytes()`

[Cohen 1981]	"On Holy Wars and a Plea for Peace"
[Harold 1997]	Chapter 2, "Primitive Data Types, Cross-Platform Issues"