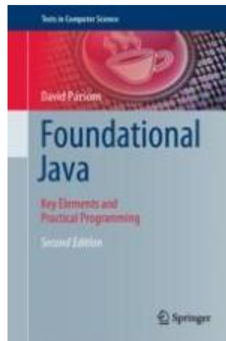


Chapter 13
Input and Output Streams

Foundational Java

Key Elements and Practical Programming

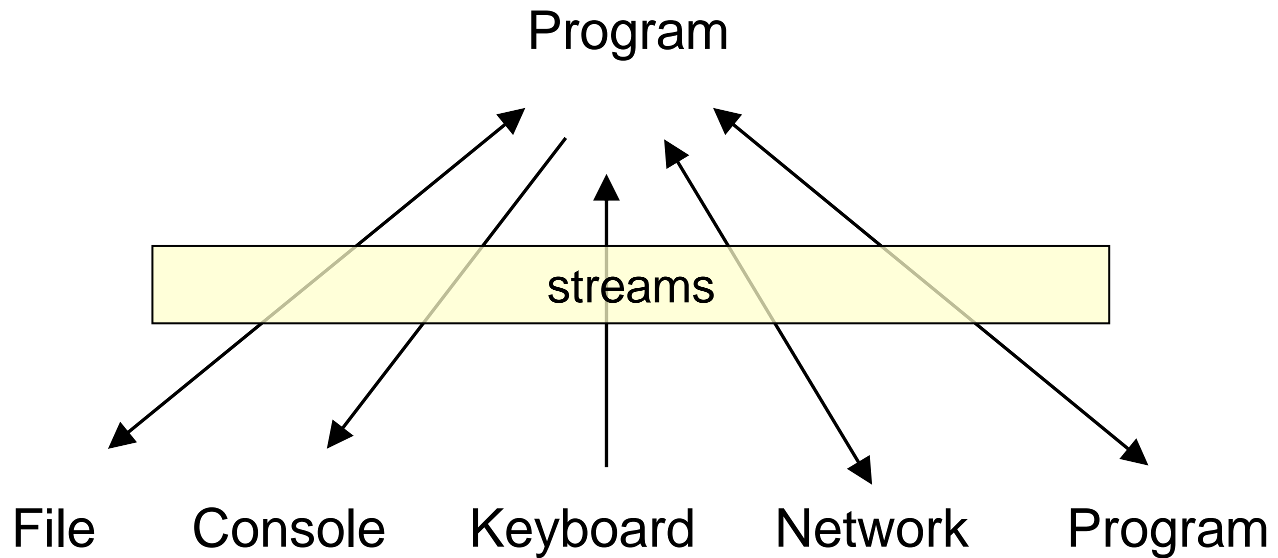
Foundational Java 2nd edition by David Parsons © 2020

Streams

- A stream is an ordered sequence of bytes flowing from a source to a destination
- Encapsulates serial output and input behind consistent abstractions
  - Regardless of source or destination
- The details specific to the source or destination are handled by the stream
  - e.g. How to write to a file versus how to write to the console

Using Streams

- The developer just has to choose the appropriate type of stream and use its methods



Java Stream Classes

- There are stream classes in both the java.io and java.io packages
- Streams (Basic I/O)
  - JDK 1.0
- Readers and Writers (Character I/O)
  - JDK 1.1
- NIO (New Input/Output) (Scaleable I/O)
  - JDK 1.4
- NIO2 (NIO enhancements)
  - Java 7

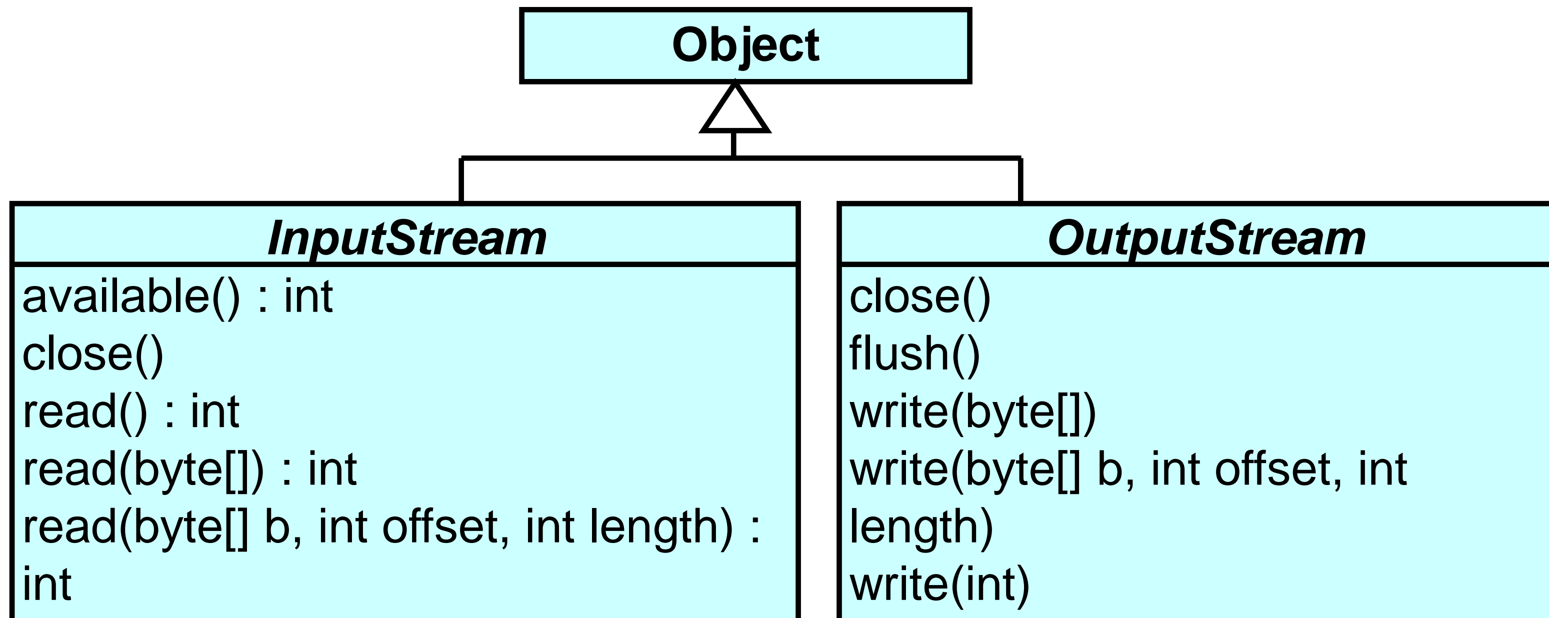
Foundational Java 2nd edition by David Parsons © 2020

Different Types of Stream

Stream type	Data types	Related Classes
Byte streams	Bytes (and arrays of bytes)	InputStream / OutputStream
Filter streams	Java primitives	DataInput / DataOutput
Character streams	Characters and Strings	Reader / Writer
Object streams	Objects	ObjectInputStream / ObjectOutputStream

Byte Streams

- InputStream and OutputStream handle bytes and byte arrays
  - These are both abstract classes with some behavior implemented by subclasses



Streaming Bytes to and From Files

- FileOutputStream is a concrete subclass of OutputStream, and can be used to write bytes to files
- The constructor shown here creates a new file

```
OutputStream outfile = new FileOutputStream("hello.txt");
```

- Alternatively we can append an existing file by passing 'true' as a second parameter argument.

```
OutputStream outfile = new FileOutputStream("hello.txt", true);
```

- 'write' methods can be used to write ASCII characters to the output stream

```
outfile.write('a');                    // ASCII chars can be bytes
```

Core Aspects of Output Streams

- java.io.IOException must be handled
- FileNotFoundException must be handled
  - After IOException
- Streams should always be closed
  - Best in a ‘finally’ block
- If an output stream is neither flushed nor closed, there is no guarantee that any data will be written

Foundational Java 2nd edition by David Parsons © 2020

FileInputStreams

- FileInputStream can be used to read bytes from files

```
InputStream infile = new FileInputStream(″hello.txt");
```

- Can read single bytes in a loop

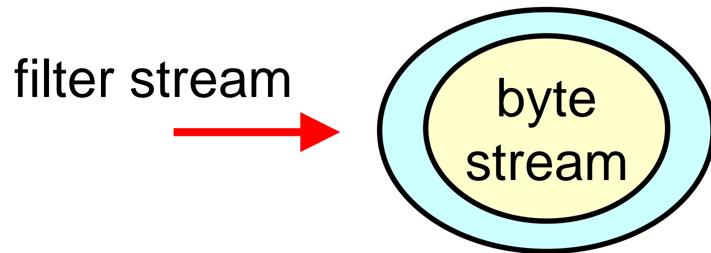
```
while(infile.available() > 0) {
		int byteValue = infile.read();
```

- Or byte arrays

```
byte[] buffer = new byte[100];
int bytesRead = infile.read(buffer);
```

Filter Streams

- You can wrap filter streams around simpler byte streams to work with Java data types

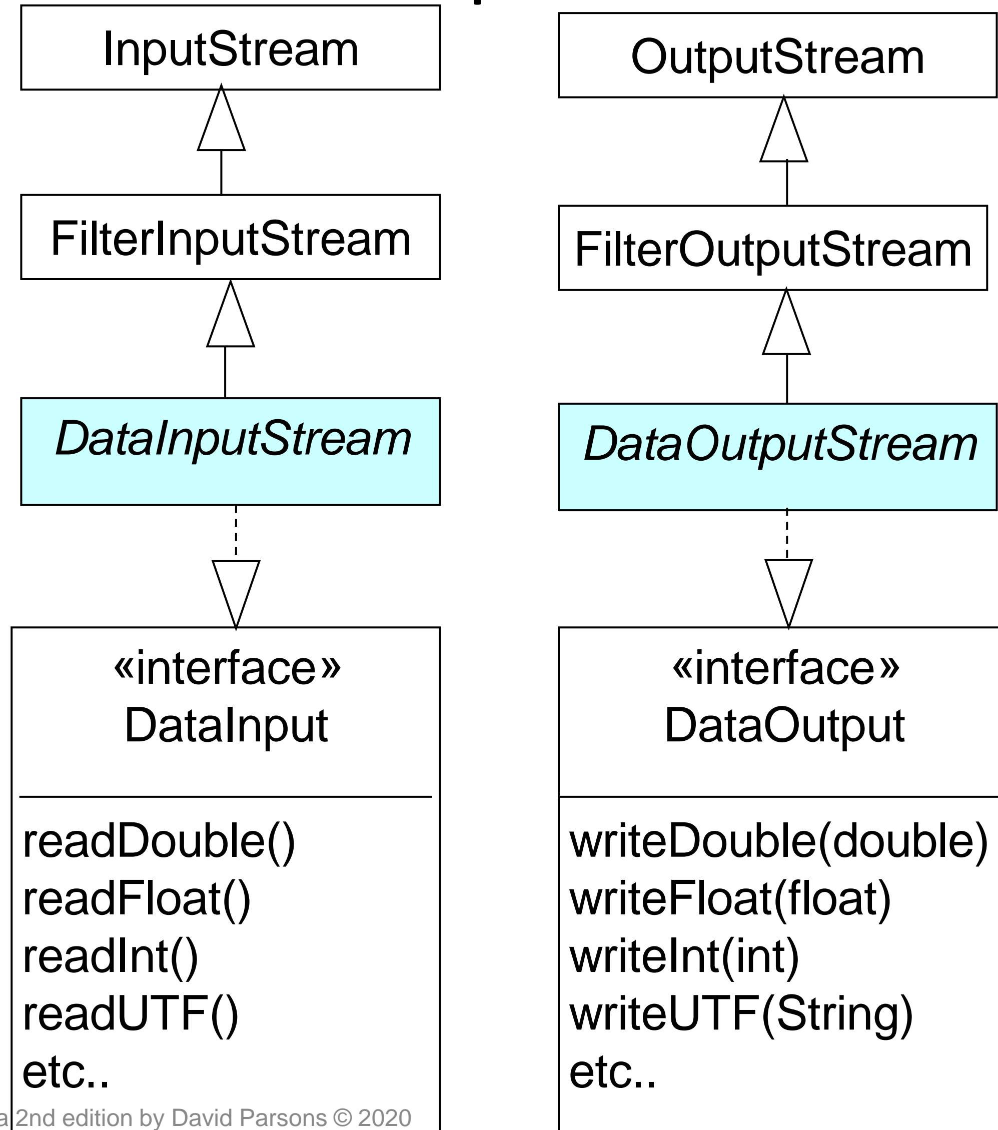


- Filter stream class constructors (such as DataOutputStream) require lower level byte streams to be passed as parameters

```
OutputStream fileOut = new FileOutputStream("data.dat");
DataOutputStream dataOut = new DataOutputStream(fileOut);
```

DataInput and DataOutput

- DataInputStream and DataOutputStream inherit from filter streams and implement interfaces that declare their read and write methods for Java primitives



Streaming Primitives

- DataOutputStreams can write Java primitives using custom ‘write’ methods

```
outfile = new FileOutputStream("data.dat");
dataOut = new DataOutputStream(outfile);
dataOut.writeUTF("Intro to Java");
dataOut.writeInt(3);
dataOut.writeDouble(1000.0)
```

- DataInputStreams have equivalent ’read’ methods

```
infile = new FileInputStream("data.dat");
dataIn = new DataInputStream(infile);
String name = dataIn.readUTF();
int numberOfDays = dataIn.readInt();
double price = dataIn.readDouble();
```

RandomAccessFile

- Combines the functionality of both DataInput and DataOutput
- Supports the methods of both interfaces
- Provides methods for managing a file pointer
  - Can be set to read or write at any position in the file
- Useful where primitive types need to be regularly both written and read

Exercise 13.1 (1)

- This table shows some data about financial transactions

Account Number	Amount	Transaction Type	Date
1009876	145.50	DR	2020-12-03
1876253	1267.00	CR	2012-11-30
1192873	45.30	CR	2012-02-15

- Create a class with a ‘main’ method that writes this data to a file
- Assume that the account number is to be written as an integer and the transaction amount as a double

Foundational Java 2nd edition by David Parsons © 2020

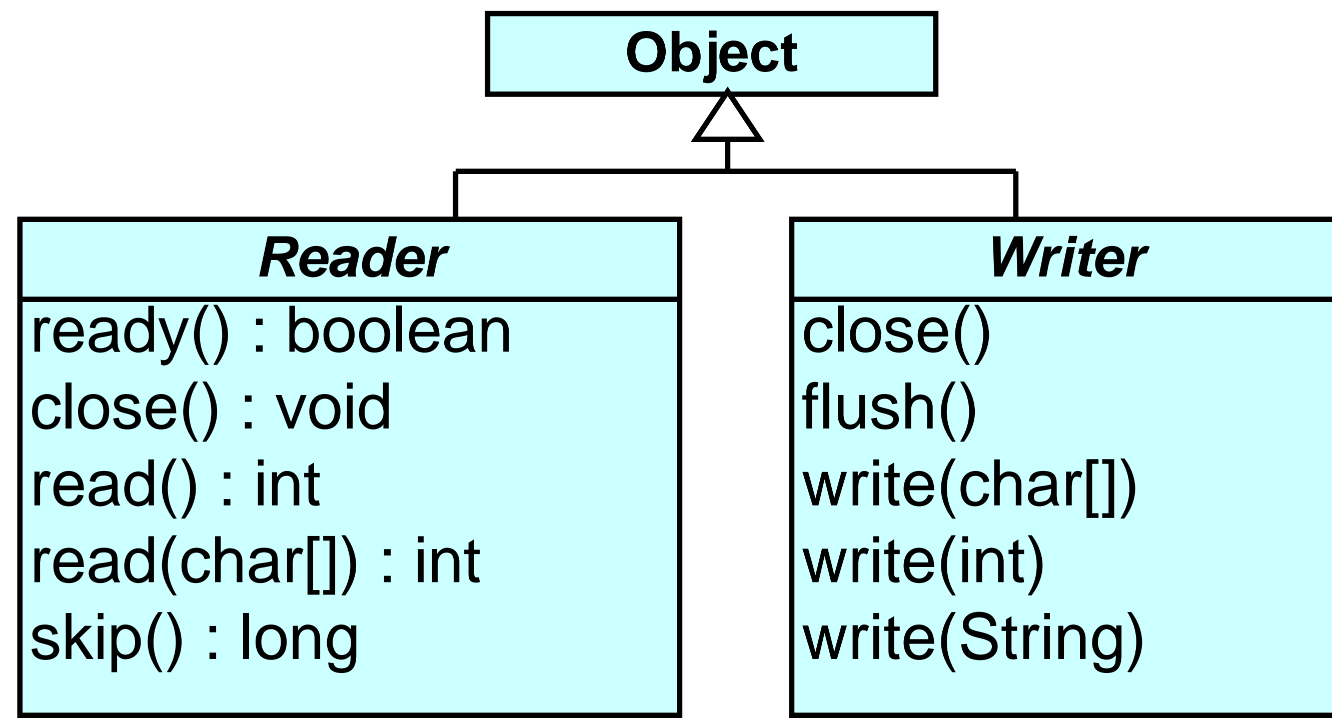
Exercise 13.1 (2)

- The transaction type could be written as a String or two separate characters
- To write a Date, use a Calendar to set the required date and use the ‘long’ data returned from the ‘getTimeInMillis’ method to write to the file
  - The dates here are shown in the format yyyy-MM-dd
- When you have successfully written the data to a file, create another class with a ‘main’ method that can read the data from the file and display it, suitably formatted, on the console

Readers and Writers

- Readers and Writers are in a separate hierarchy from the InputStream and OutputStream classes.
- They have a similar set of methods but handle chars and Strings, not bytes
  - These are both abstract classes with some behaviour implemented by subclasses
  - The preferred streams to use when reading and writing character data

Reader and Writer Classes



BufferedReader and BufferedWriter

- For efficiency, use BufferedReader and BufferedWriter for character data
  - Concrete subclasses of Reader and Writer
- Use other concrete subclasses as part of their implementation
  - e.g. BufferedWriter constructor requires a Writer object to be passed as a parameter argument

```
FileWriter fileWriter = null;
BufferedWriter writer = null;
try {
  fileWriter = new FileWriter("characterfile.txt");
  writer = new BufferedWriter(fileWriter);
```

BufferedWriter

- The ‘write’ method can be used to write Unicode characters or Strings to the output stream
- The ‘newLine’ method provides a platform independent way of writing a new line character.

```
writer.write("I am the first line of the file");
writer.newLine();
writer.write("I am the second line of the file");
writer.newLine();
writer.flush();
```

BufferedReader

- The ‘read’ method can read a single Unicode character from the input stream
- The ‘readLine’ method can read lines of text into Strings

```
FileReader fileReader = null;
BufferedReader textReader = null;
String textLine = null;
try {
   fileReader = new FileReader("characterfile.txt");
   textReader = new BufferedReader(fileReader);
   while (textReader.ready()) {
    textLine = textReader.readLine();
    System.out.println(textLine);
  }
```

PrintStreams

- System.out and System.err are instances of PrintStream
- Subclass of FilterStream
  - PrintStream has overloaded print() and println() methods to output primitive types and strings
- Both normally print to the console
- Can use PrintStreams for other types of output

PrintWriters

- Similar to PrintStream, but is a subclass of Writer
- Can use the services of a Writer object
- Should be used in preference to a PrintStream when the output is character-based
  - e.g. writing text to a file

```
output = new PrintWriter(new FileWriter("story.txt"));
output.println("My Story");
output.print("Chapter ");
output.println(1);
output.print("One upon a Time, in fact " + new Date() + "\nThe End");
```

The File Class

- Can be used to represent files in the operating system
  - Can also be used to interact with the file system
  - Replaced by 'Path' in Java 7
- Overloaded constructors:

```
File f1 = new File("/temp/temp.txt");          // path and filename as one parameter
```

  - or

```
File f1 = new File("/temp","temp.txt");        // path and filename as separate parameters
```

- A File object can be used to construct a file stream object

```
FileInputStream infile = new FileInputStream(f1);
```

Platform Independent File Paths

- To work across different platforms, your file paths cannot include platform-specific separators
  - e.g '\' ':' and '/'
- You can build platform-independent paths using the static 'separator' field of the File class

```
FileInputStream infile = null;
String sep = File.separator;
File file = new File(sep + "data" + sep + "hello.txt");
if (file.exists())
{
    infile = new FileInputStream(file);
//...
}
```

Streaming Objects

- You can read and write objects using ObjectInputStreams and ObjectOutputStreams
  - It wraps an OutputStream
  - It has a writeObject() method to stream an object, e.g. a Date

```
Date myDate = new Date();
ObjectOutputStream out = new ObjectOutputStream
                (new FileOutputStream("date.dat"));
out.writeObject(myDate);
```

```
ObjectInputStream in = new ObjectInputStream
                (new FileInputStream("date.dat"));
Date newDate = (Date)in.readObject();
```

- You can only stream objects that implement the Serializable interface

```
Class MyClass implements java.io.Serializable
```

Serializing Course Objects

- To serialize a Course object (and its Modules) both classes must implement the java.io.Serializable interface

```
public class Course implements Serializable
{
// etc.
```

```
public class Module implements Serializable
{
// etc.
```

Exercise 13.2

- Create a Transaction class that has the fields defined in the table from Exercise 13.1, along with their getters and setters
- Make the class implement the Serializable interface
- Override ‘toString’ to provide a readable String representation of a Transaction
- Write a JUnit test that writes a Transaction object to a file and then reads it back again, testing that the state of the object has been correctly restored from the file
- Can you write a JUnit test for the toString method?

The New IO Library

- Multiplexed, Non-Blocking, I/O facility
  - This means that it can blend many different streams (channels) together into one
  - And that each of the channels may stream data at rates independent of the others
  - Comprises buffers, channels, selectors and charsets

The Main Types in NIO Class Library

Class Types	Characteristics
Buffers	Containers for data. Concrete subclasses for different data types
Charsets (and their associated decoders and encoders)	Translate between bytes and Unicode characters
Channels	Represent connections to entities capable of performing I/O operations
Selectors and selection keys	Together with selectable channels define a multiplexed, non-blocking I/O facility

Buffer Classes

- Extend the abstract class java.nio.Buffer
- e.g.: ByteBuffer, CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer, LongBuffer, ShortBuffer
- Common methods
  - put
  - get
- Tailored to the data types that they handle
- Defined in the various subclasses of Buffer

Foundational Java 2nd edition by David Parsons © 2020

Invariants

0 <= *mark* <= *position* <= *limit* <= *capacity*

- *capacity*
  – The number of elements in the buffer
- *limit*
  – Stores the index of the first element from which the buffer should not be accessed
- *position*
  – The index of the next element to be read or written
- mark
  – Calling the ‘reset’ method will move the current position to the mark

Foundational Java 2nd edition by David Parsons © 2020

Methods Associated With Invariants

- clear()
  - Clears the buffer ready for operations to put data
  - Sets the limit to the capacity and the position to zero
- flip()
  - Makes the buffer ready for operations to get data
  - Sets the limit to the current position and the position to zero
- rewind()
  - Makes a buffer ready for re-reading the data
  - The limit is unchanged but the position is set to zero

ByteBuffers

- We read and write data using ByteBuffers
- These have to be allocated before use using a factory method

```
ByteBuffer buffer = ByteBuffer.allocate(1024);
```

- The ‘allocate’ method returns a *non-direct* ByteBuffer
- ‘allocateDirect’ returns a *direct* ByteBuffer

```
ByteBuffer buffer = ByteBuffer.allocateDirect(1024);
```

‘put’ and ‘get’ Methods

- Primitives can be added using putInt, putChar etc.

```
buffer.putInt(1);
```

- The buffer needs to be flipped between reading and writing (resets the position to the beginning)

```
buffer.flip();
```

- Then we can get values from the buffer, using the appropriate ‘get’ methods e.g.

```
int myInt = buffer.getInt();
```

File Channels

- Channel is an interface implemented by a number of different concrete channel classes
- e.g. FileChannel java.nio.channels.FileChannel
- Safe for use with multiple threads
  - Created from a file stream object

```
FileOutputStream fileout = new FileOutputStream("C:/myfile.txt");
FileChannel fc = fileout.getChannel();
```

- Data is written from a ByteBuffer

```
fc.write(buffer);
```

Reading From a Channel

- We need to read into an allocated buffer
- Flip the buffer before getting data
- There are ‘get’ methods for different data types

```
ByteBuffer buffer = ByteBuffer.allocate(1024);
try
{
  FileInputStream filein = new FileInputStream("myintfile.dat");
  FileChannel fc = filein.getChannel();
  fc.read(buffer);
  buffer.flip();
  while(buffer.hasRemaining())
  {
    System.out.println(buffer.getInt());
  }
```

View Buffers

- Instead of reading data from a ByteBuffer we can create buffers of the appropriate type
  - e.g. IntBuffer
- ‘as’ factory methods return views for different data types

```
IntBuffer intBuf = buffer.asIntBuffer();
```

- Then the various ‘get’ methods work with the actual type

```
int num1 = intBuf.get();
```

Charsets and String Data

- NIO works a lot at the ByteBuffer level
  - Charsets and CharBuffers let us work with Strings
  - A Charset is created for a specific character encoding
  - Has factory method to create encoders and decoders for moving character data between CharBuffers and ByteBuffers

```
Charset charset = Charset.forName("utf-8");
CharsetEncoder encoder = charset.newEncoder();
ByteBuffer buffer = null;
try {
  buffer = encoder.encode(CharBuffer.wrap("a string\nanother string"));
}
catch (CharacterCodingException e) {
  e.printStackTrace();
}
```

Decoding Characters

- Read the data into aByteBuffer
- Flip the ByteBuffer and decode

```
Charset charset = Charset.forName("utf-8");
CharsetDecoder decoder = charset.newDecoder();
ByteBuffer buffer = ByteBuffer.allocate(1024);
try
{
  FileInputStream filein = new  FileInputStream("myencodedfile.txt");
  FileChannel fc = filein.getChannel();
  fc.read(buffer);
  buffer.flip();
  CharBuffer cbuf = decoder.decode(buffer);
  String s = cbuf.toString();
  System.out.println(s);
```

Exercise 13.3

- Create a StringFileHandler class
- Add a static method to write a String to a given file using appropriate classes from the java.nio package and its sub packages
- Add another static method to read a String from a given file
- Test your StringFileHandler class methods

Summary

- Different types of stream handle the input and output of bytes, primitives, Strings and objects
- Lower level stream classes support the implementation of higher level steams by being passed to their constructors
- Buffers, Charsets and Channels in the java.nio package

Foundational Java 2nd edition by David Parsons © 2020