Reading Characters

José Paumard
PHD, JAVA CHAMPION, JAVA ROCK STAR
@JosePaumard https://github.com/JosePaumard

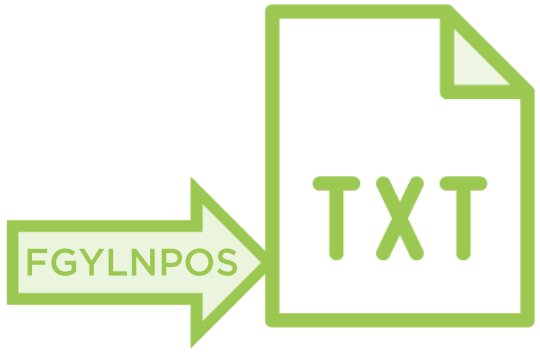
Agenda

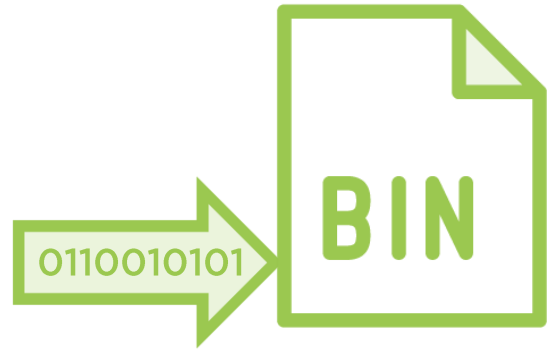
- Concept of Reader
- How to read characters from files
- How to read files line by line
- Then from in-memory arrays
- How to use character sets

Introducing Readers


TXT
FGYLNPOS


BIN
0110010101


FGYLNPOS
TXT


0110010101
BIN

The Reader Abstract Class

The Reader is an abstract class

It defines the basic operations:

- Reading of a single character
- Reading of an array of characters
- Marking and resetting a given position
- Skipping positions

And it can be closed

```
Reader reader = ...; // we will see how to create a reader later

int nextChar = reader.read();

while (nextChar != -1) {
    // do something with nextChar
    nextChar = reader.read();
}
```

When there are no more characters to read, the `read()` call returns -1

“Do something” might be to store the characters in a buffer

```java
Reader reader = ...;
char[] buffer = new char[1024];


int number = reader.read(buffer);

while (number != -1) {
    // do something with buffer
    number = reader.read(buffer);
}
```

When there are no more characters to read, the `read()` call returns -1

Be careful, `number` is the number of characters that have been read

It can be less than 1024

```
Reader reader = ...;
char[] buffer = new char[1024];


int number = reader.read(buffer, 16, 128);

while (number != -1) {
    // do something with buffer
    number = reader.read(buffer, 16, 128);
}
```

When there is no more characters to read, the `read()` call returns -1

Be careful, `number` is the number of characters that have been red

It can be less than 1024

Dealing with Exceptions

I/O Operations Will Throw Exceptions

All these methods declare checked exceptions

We need to add some code to be executed in case something goes wrong...

A disk that is not there, a network resource that is unavailable

```
Reader reader = ...;                    // probably throws an exception

int nextChar = reader.read();           // throws IOException

while (nextChar != -1) {
    // do something with nextChar
    nextChar = reader.read();           // throws IOException
}
```

As a rule of thumb, all the method calls may throw an `IOException`

Two strategies to handle a checked exception:

- to throw it to the caller

- to handle it locally

In both cases, log it somewhere!

```
Reader reader = null;

try {
    reader = ...;

    int nextChar = reader.read();

    while (nextChar != -1) {
        // do something with nextChar
        nextChar = reader.read();
    }
} catch (IOException e) {
    // deal with the exception
}
```

Closing a Reader

System Resources Must Be Closed

A Reader uses a system resource

As such, it must be properly closed

There are two patterns for that:

- call the close() method

- use the try-with-resource pattern

Closing a resource is tricky!

```
Reader reader = null;

try {
    reader = ...;

    int nextChar = reader.read();

    while (nextChar != -1) {
        // do something with nextChar
        nextChar = reader.read();
    }
    reader.close();
} catch (IOException e) {
    // deal with the exception
}
```

Problem: this code is buggy!

Why?

```
Reader reader = null;

try {
    reader = ...;

    int nextChar = reader.read();

    while (nextChar != -1) {
        // do something with nextChar
        nextChar = reader.read();
    }
    reader.close();
} catch (IOException e) {
    // deal with the exception
}
```

- Suppose something goes wrong here
- Or here

```
Reader reader = null;

try {
    reader = ...;

    int nextChar = reader.read();

    while (nextChar != -1) {
        // do something with nextChar
        nextChar = reader.read();
    }
    reader.close();
} catch (IOException e) {
    // deal with the exception
}
```

- The execution is interrupted here
- Or here
- And continues there

```
Reader reader = null;

try {
    reader = ...;

    int nextChar = reader.read();

    while (nextChar != -1) {
        // do something with nextChar
        nextChar = reader.read();
    }
    reader.close();
} catch (IOException e) {
    // deal with the exception
}
```

- And the close() method is never called...

```
Reader reader = null;

try {
    reader = ...;
    // do something with reader

} catch (IOException e) {
    // deal with the exception

} finally {
    reader.close();
}
```

```
Reader reader = null;

try {
    reader = ...;
    // do something with reader

} catch (IOException e) {
    // deal with the exception

} finally {
    try {
        reader.close();
    } catch (IOException e) {
        // deal with the exception
    }
}
```

```
Reader reader = null;

try {
    reader = ...;                          // if an exception is thrown
    // do something with reader            // here

} catch (IOException e) {
    // deal with the exception

} finally {
    try {
        reader.close();                    // then a NullPointerException
    } catch (IOException e) {              // is thrown here...
        // deal with the exception
    }
}
```

```
Reader reader = null;

try {
    reader = ...;
    // do something with reader

} catch (IOException e) {
    // deal with the exception

} finally {
    if (reader != null) {
        try {
            reader.close();
        } catch (IOException e) {
            // deal with the exception
        }
    }
}
```

Writing a correct code like this one is tedious and error prone...

Fortunately it has been improved in Java 7

With the try-with-resources pattern

```
try (Reader reader = ...;) {
    // do something with reader
} catch (IOException e) {
    // deal with the exception
}
```

Much easier to write and read

Several resources can be opened in the `try()`

They will be closed automatically when leaving the try block

```
try (Reader reader = ...;) {
    // do something with reader
} catch (IOException e) {
    // deal with the exception
}
```

To be used in this pattern, a resource must implement `AutoCloseable`

With only one method to implement: `close()`

Very easy to use for homemade resources

Marking, Resetting and Skipping

A Reader can skip elements

- Supported by all readers

A Reader may support reset

- One cannot test if is does or not

A Reader may support mark

- Testable with markSupported()

A mark() call puts a flag on a given element

A reset() call rewinds to the previously marked element, or the beginning of the stream

A skip() call skips the next elements

```
Reader reader = ...;

reader.mark(1024);

// reader at most 1024 chars

reader.reset();
```

The mark method takes the number of chars that can be read before the mark becomes invalid

Creating Readers

Two Ways of Extending a Reader

Reminder: Reader is an abstract class

Extended by 2 categories of concrete classes

1) classes for a certain type of input

- Disk: FileReader

- In-memory: CharArrayReader, StringReader

2) classes that add behavior to Reader

- BufferedReader
- LineNumberReader
- PushbackReader

```
File file = new File("files/data.txt");

Reader reader = new FileReader(file);
```

The `FileReader` class creates a reader on a file

```
String text = "Hello world!";

Reader reader = new StringReader(text);
```

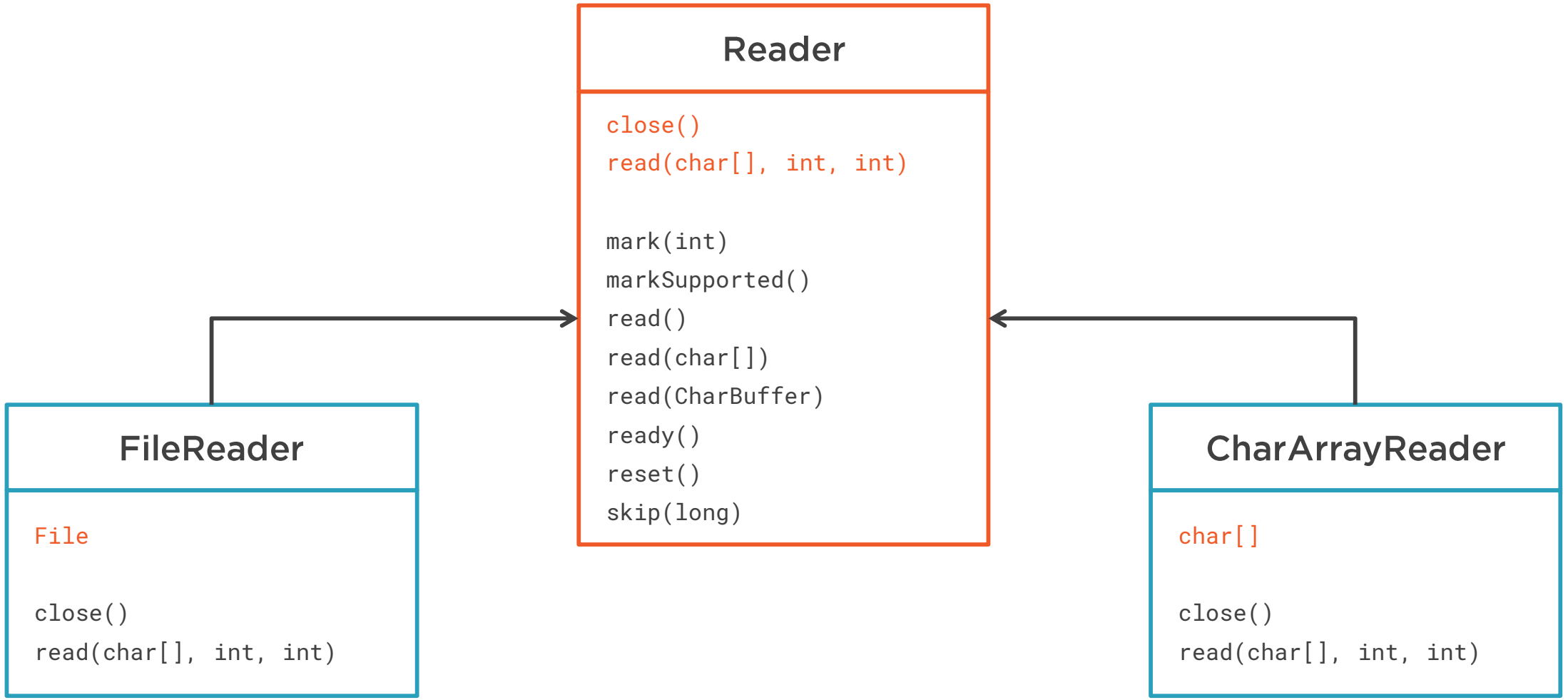
The `FileReader` class creates a reader on a file

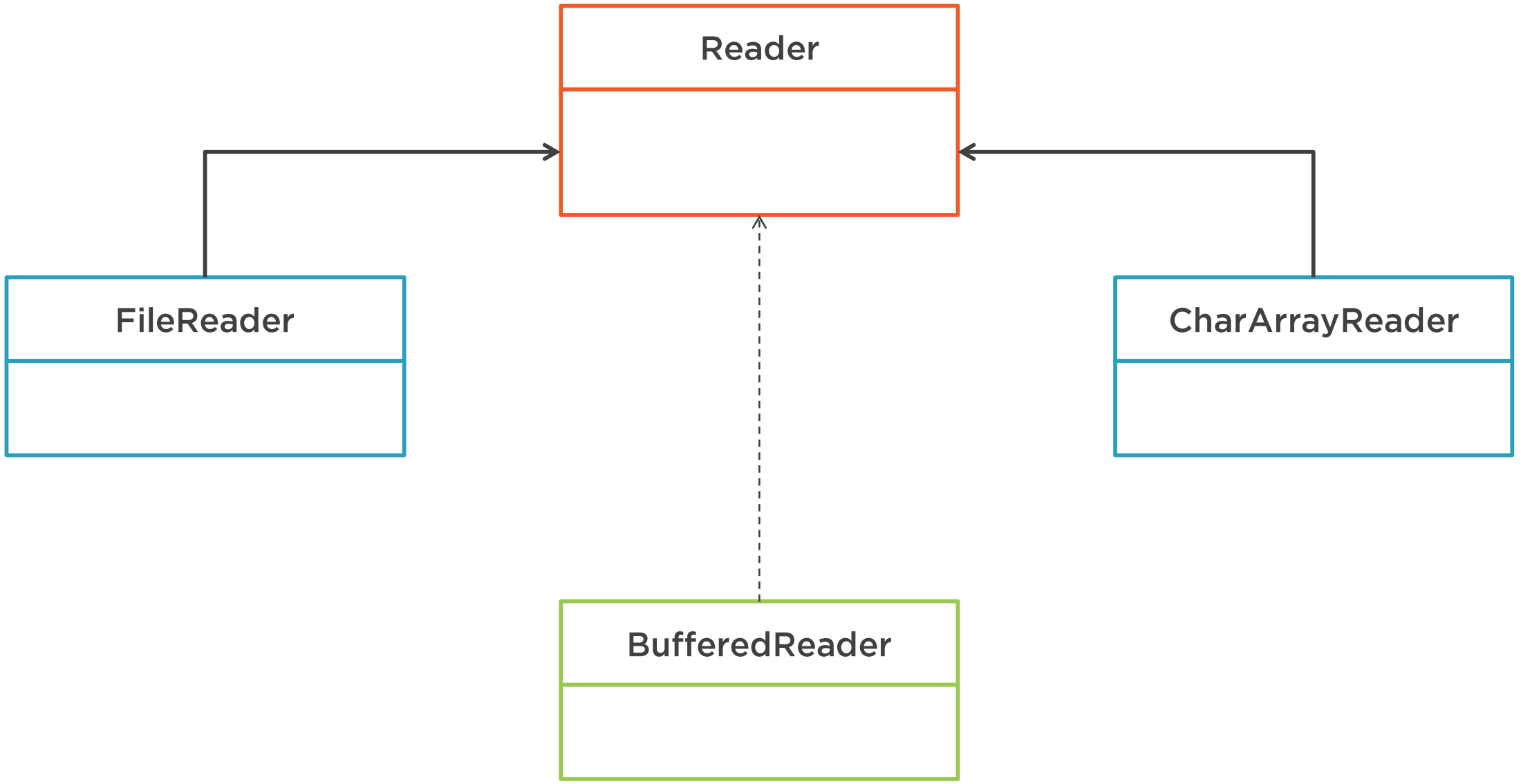
One can also open a reader on an in-memory array or a string

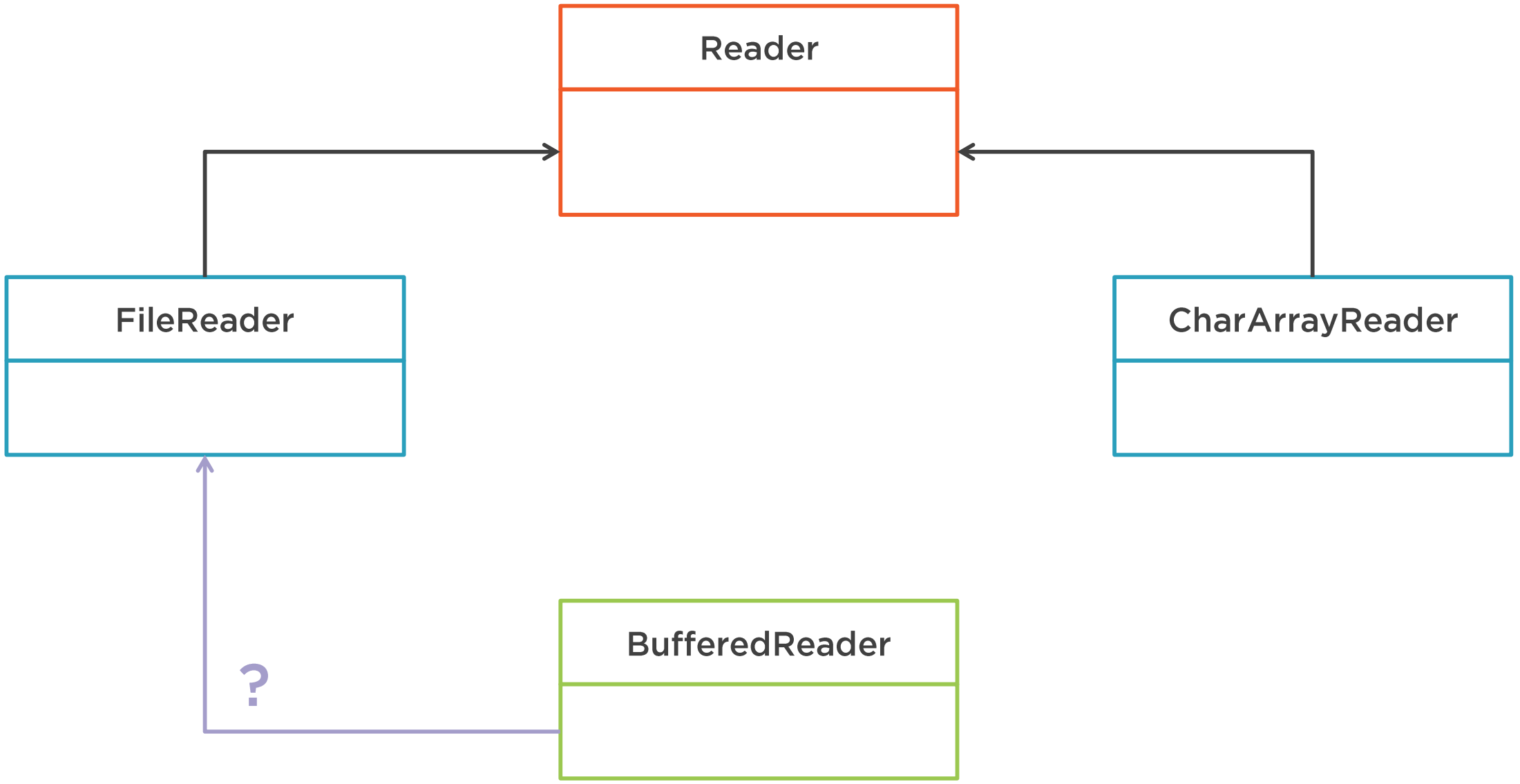
Behavior extensions follow the GoF Decorator pattern

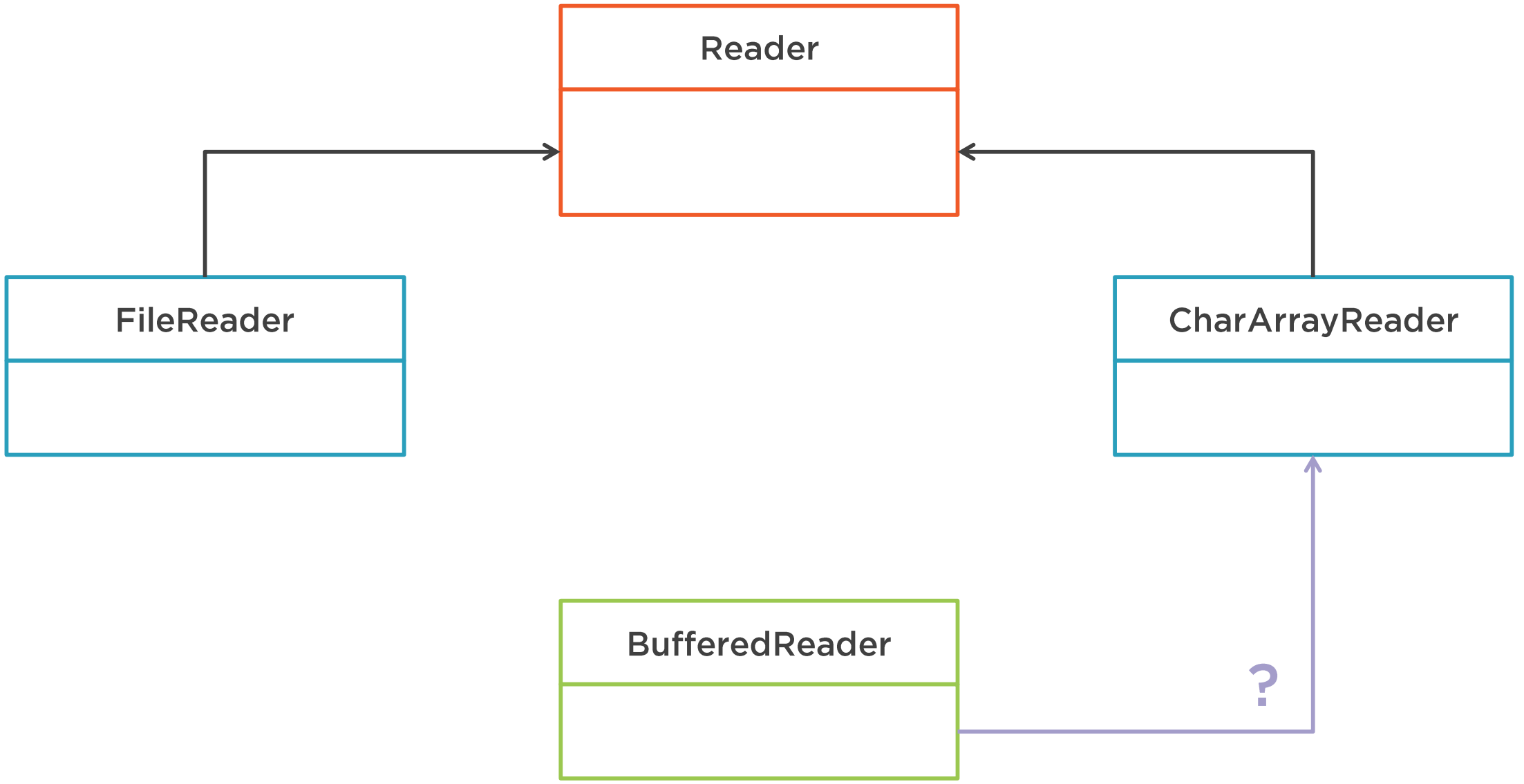
BufferedReader extends Reader

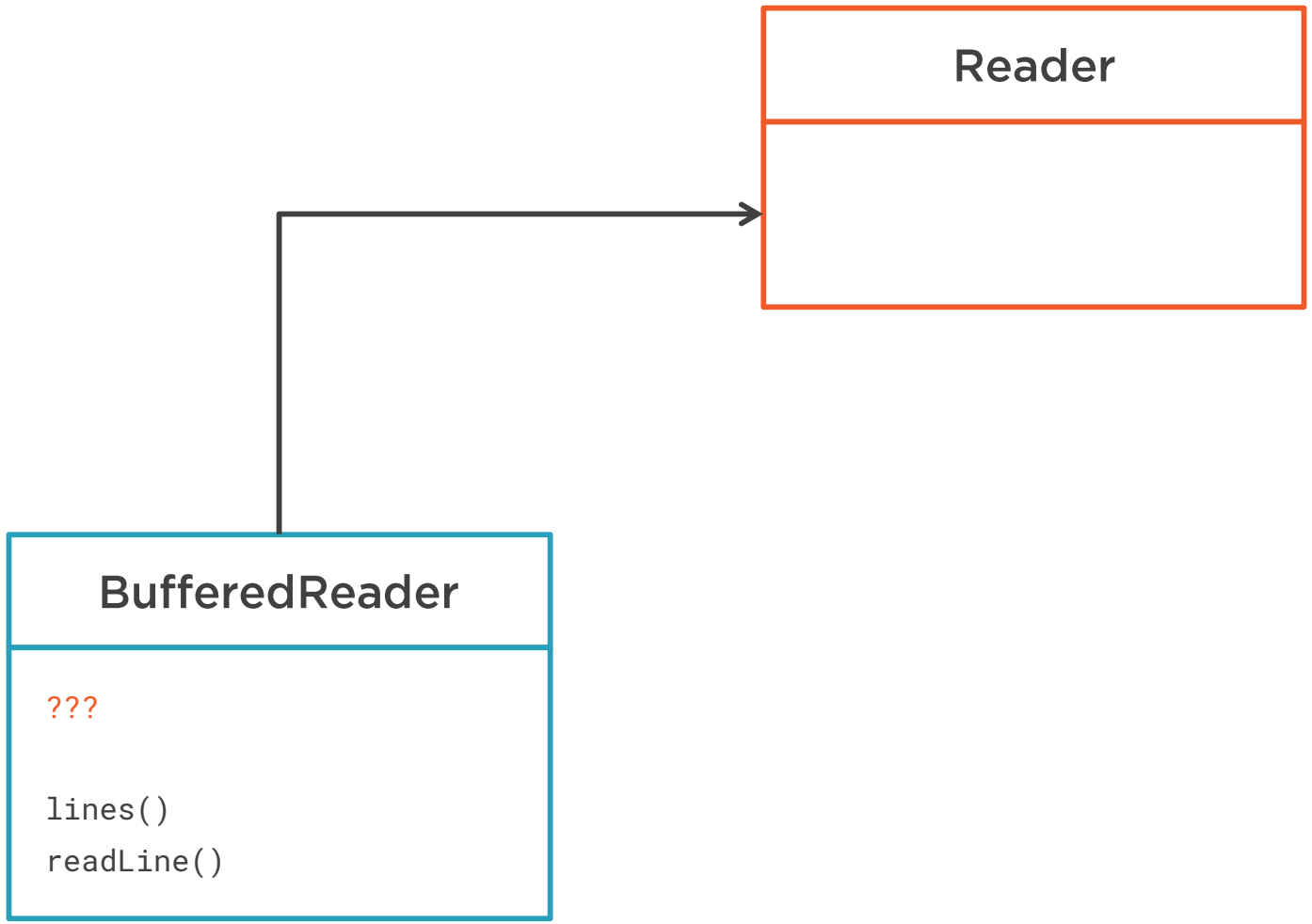
And is built on an instance of Reader

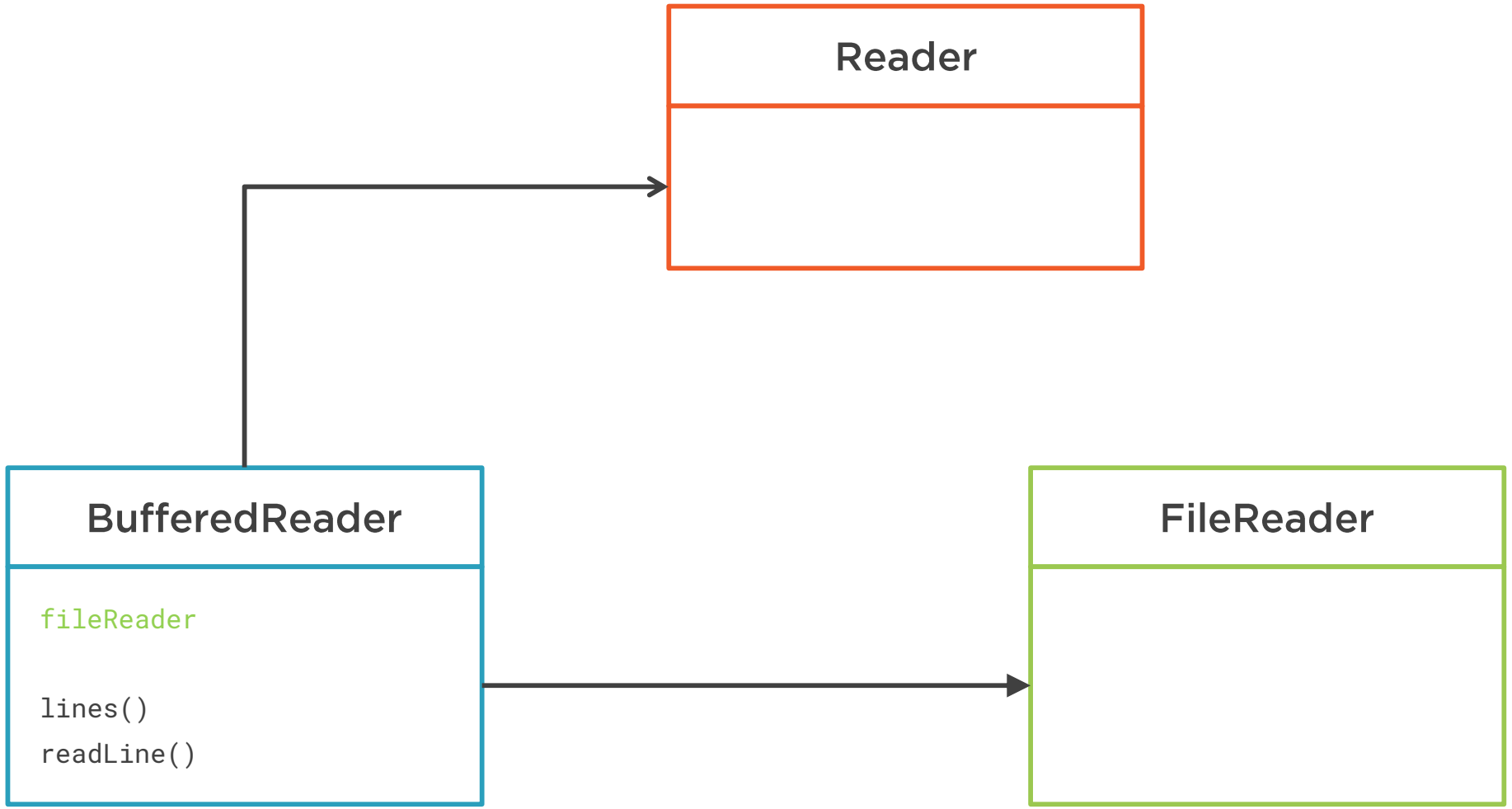

Reader
close()
read(char[], int, int)
mark(int)
markSupported()
read()
read(char[])
read(CharBuffer)
ready()
reset()
skip(long)
FileReader
File
close()
read(char[], int, int)
CharArrayReader
char[]
close()
read(char[], int, int)

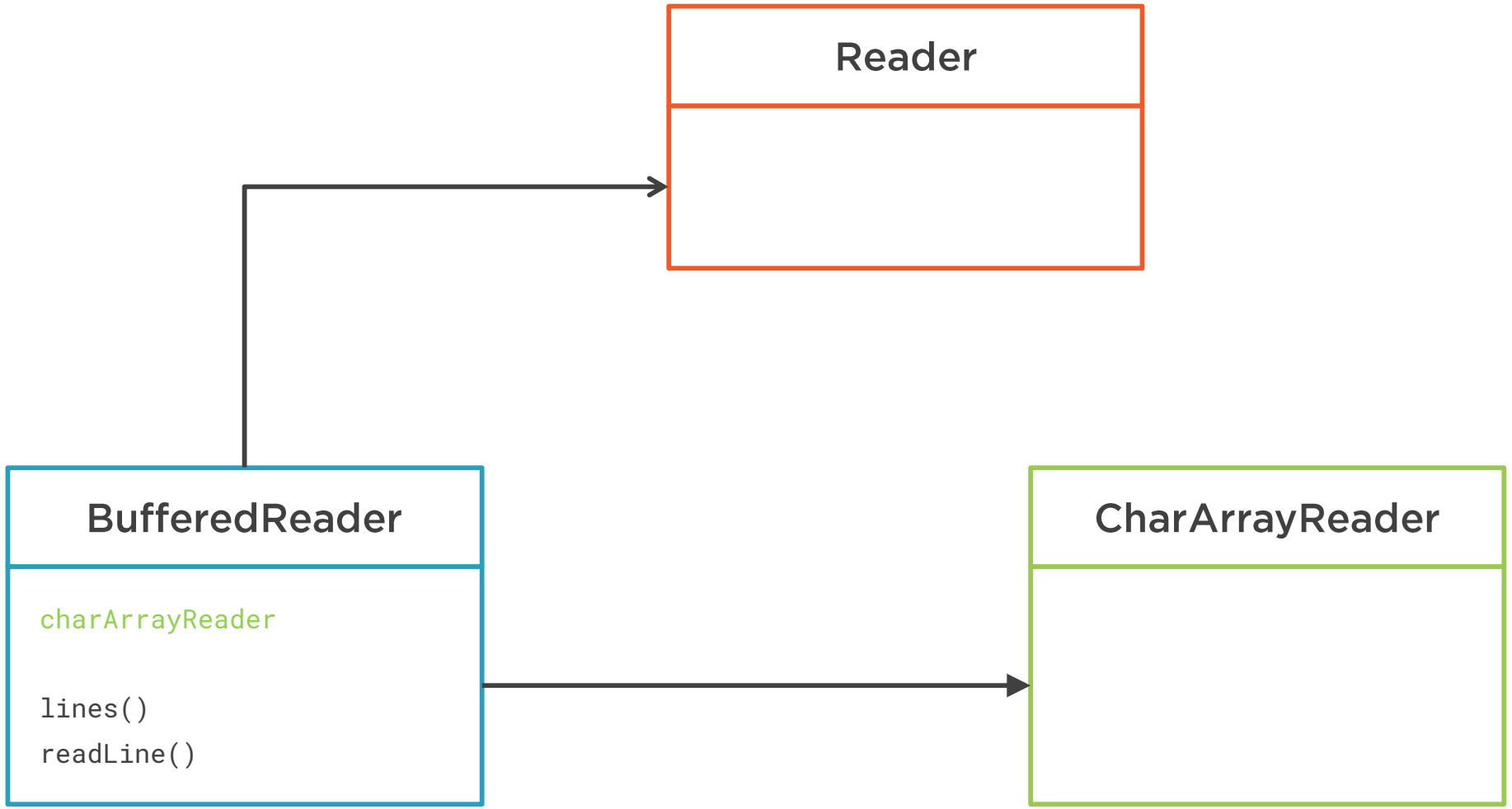

Reader
FileReader
CharArrayReader
BufferedReader


Reader
FileReader
CharArrayReader
?
BufferedReader


Reader
FileReader
CharArrayReader
BufferedReader
?


Reader
BufferedReader
???
lines()
readLine()


Reader
BufferedReader
fileReader
lines()
readLine()
FileReader


Reader
BufferedReader
charArrayReader
lines()
readLine()
CharArrayReader

```
File file = new File("files/data.txt");

FileReader fileReader = new FileReader(file);

BufferedReader bufferedReader = new BufferedReader(fileReader);
```

Let us create a `BufferedReader`

Reads the chars through a buffer

`BufferedReader` adds the `readLine()` method to Reader

And also supports the mark and skip operations

```
File file = new File("files/data.txt");

FileReader fileReader = new FileReader(file);

LineNumberReader lineNumberReader = new LineNumberReader(fileReader);
```

Same pattern for `LineNumberReader`

`LineNumberReader` **extends** `BufferedReader`

It adds a `getLineNumber()` **method**

Creating concrete readers using the constructor are Java 1 patterns

Java 7 introduced factory methods

```
File file = new File("files/data.txt");
FileReader fileReader = new FileReader(file);
BufferedReader bufferedReader = new BufferedReader(fileReader);

Path path = Paths.get("files/data.txt");
BufferedReader reader2 =
    Files.newBufferedReader(path);
```

In this case, the file is read with the UTF-8 charset

```java
File file = new File("files/data.txt");
FileReader fileReader = new FileReader(file);
BufferedReader bufferedReader = new BufferedReader(fileReader);

Path path = Paths.get("files/data.txt");
BufferedReader reader2 =
   Files.newBufferedReader(path, StandardCharsets.ISO_8859_1);
```

In this case, the file is read with the UTF-8 charset

But one can also pass other charsets

For the record: this reader is built on an `InputStreamReader`

Closing Decorated Readers

```
File file = new File("files/data.txt");

FileReader fileReader = new FileReader(file);

BufferedReader bufferedReader = new BufferedReader(fileReader);
```

One last note: in this code we have opened 2 readers

```
File file = new File("files/data.txt");

FileReader fileReader = new FileReader(file);

BufferedReader bufferedReader = new BufferedReader(fileReader);
```

One last note: in this code we have opened 2 readers

The file reader

```
File file = new File("files/data.txt");

FileReader fileReader = new FileReader(file);

BufferedReader bufferedReader = new BufferedReader(fileReader);
```

One last note: in this code we have opened 2 readers

The file reader

And the buffered reader

```
File file = new File("files/data.txt");

FileReader fileReader = new FileReader(file);

BufferedReader bufferedReader = new BufferedReader(fileReader);
```

1) they are connected to the same file on the disk, so reading from one is the same as reading from the other

2) if we read from one, then from the other, there is no reset

3) closing one will close the other, no need to close both

Demo

Let us see some code!

Let us create simple readers and see them in action

See how exceptions work

And how to deal with CharSet

Module Wrap Up

What did you learn?

Readers!

Patterns to create readers, Java 1 and 7

How to open and close readers

How to deal with exceptions and charsets