Exceptions and Data Streams in Java (Files)

Dr. Eliahu Khalastchi

2017

המסלול האקדמי
המכללה למינהל

Exception Handling

- All Java programs which deal with files and/or streams
  - must include exception handling

- Exceptions are error conditions encountered in executing class methods
  - Attempting to read past an end of file
  - Attempting to read a file that doesn’t exist
  - Trying to open a malformed URL
  - Divide by zero
  - Taking the square root of a negative number
  - Etc.

Exception Handling

- Normal error handling (in C) would return an error code (eg. –1)

```
int func(){
	//...
	if(y==0)
		return -1;
	z=x/y;
	//...
}
```

- This is simple, and sometimes effective method, **but**:
- Sometimes impossible to return a valid error code
- Not object oriented!
  - no information about the error is contained in the error code
- The code gets ‘polluted’ with error checking code
- The client is not obligated to check the error code
- It is nicer to have all of the error handling in one place

Throwing Exceptions

- The method must declare the type of the thrown object
- The thrown object should store information about the error

```java
public class PersonalDetails {

 String email;

 public void setEmail(String email) throws Exception{

   if(!email.contains("@"))
      throw new Exception("not a valid address");
   this.email=email;
 }
//...
}
```

Handling Exceptions

- The calling method should either catch the exception
- Or throw it on…

```
public class Person {

 PersonalDetails pd;

 public void fillDetailsForm() throws Exception {
   //...
   String email="abc.gmail.com";
   pd.setEmail(email);
   // the following lines will be skipped
 }
}
```

Handling Exceptions

- The calling method should either catch the exception
- Or throw it on…

```
public void fillDetailsForm(){
    //...
    String email="abc.gmail.com";
    try {
          pd.setEmail(email);
          // the following lines will be skipped
    } catch (Exception e) {
         // this is called
         e.printStackTrace();
    }
                                  here...
```

```
java.lang.Exception: not a valid address
at course.java.test.PersonalDetails.setEmail(PersonalDetails.java:8)
at course.java.test.Person.fillDetailsForm(Person.java:10)
at course.java.test.Person.main(Person.java:19)
```

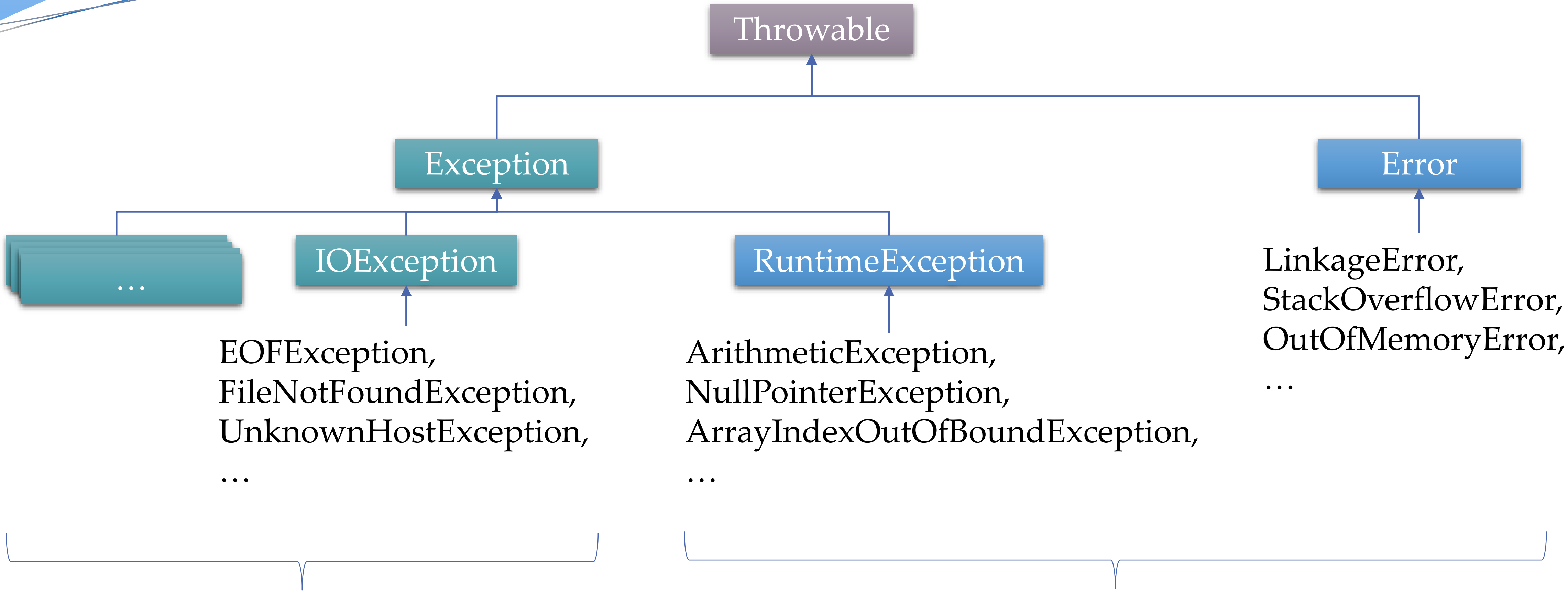
Finally block

- Always executed… use it for safe exit

```
public void fillDetailsForm(){
 String email="abc.gmail.com";
 try {
      pd.setEmail(email);
      System.out.println("this will not be printed");
 } catch (Exception e) {
      System.out.println("catching...");
      return; // exit the method
      // but not before the finally code block!
 } finally{
      System.out.println("safe exit");
 }
 // and the code will not continue here...
 System.out.println("this will not be printed");
}
```

Exception inheritance hierarchy

- Exceptions are objects encapsulating information
  - about the type of error which caused the exception
- All exception objects derive from Throwable
- Two main types of exception object
  - *RuntimeException*
    - Usually a programmer error such as divide by zero
      or trying to access an array beyond its limits
  - *IOException*
    - Not generally caused by programmer error and generally relating to I/O or network errors



Throwable

Exception

Error

…

IOException

RuntimeException

LinkageError,
StackOverflowError,
OutOfMemoryError,
…

EOFException,
FileNotFoundException,
UnknownHostException,
…

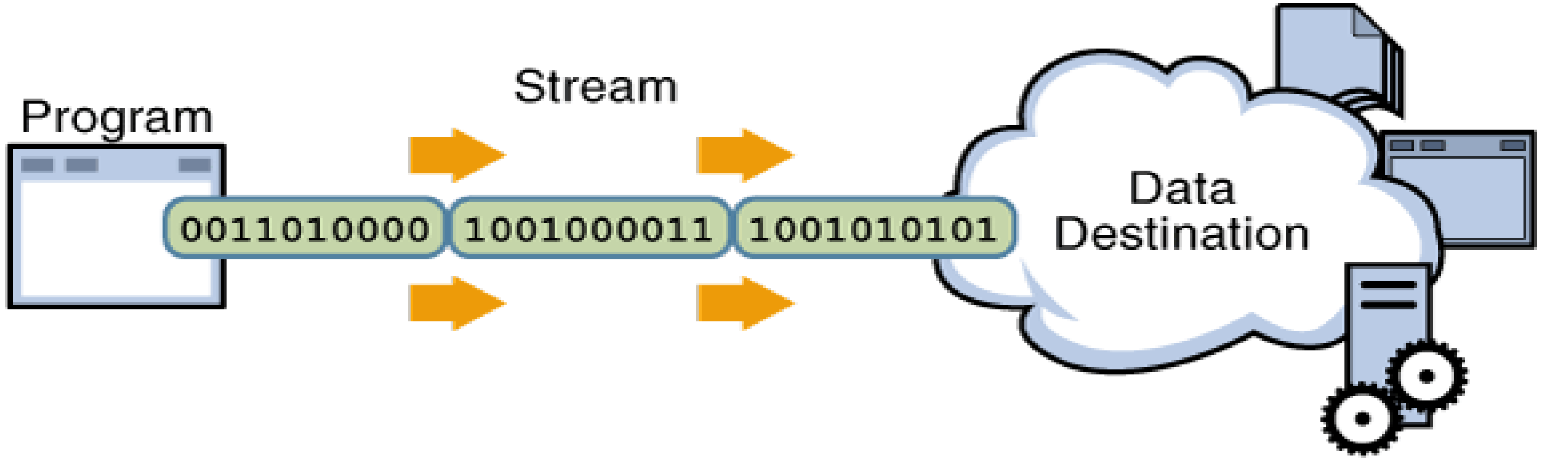
ArithmeticException,
NullPointerException,
ArrayIndexOutOfBoundException,
…

Checked Exceptions:
- are checked by the compiler to potentially occur
- you have to handle them in your code

Unchecked Exceptions:
- cannot be checked by the compiler to potentially occur
- you don’t have to handle them in your code

Data Streams in Java

Reading and writing to



input & output streams

- Most applications require data to be stored in files
  - Spreadsheets
  - Word processing
  - Etc
- Java has extensive facilities for handling files of various types
- Associated with files is the general concept of streams
- which can be applied to both **files** and **networks**

input & output streams

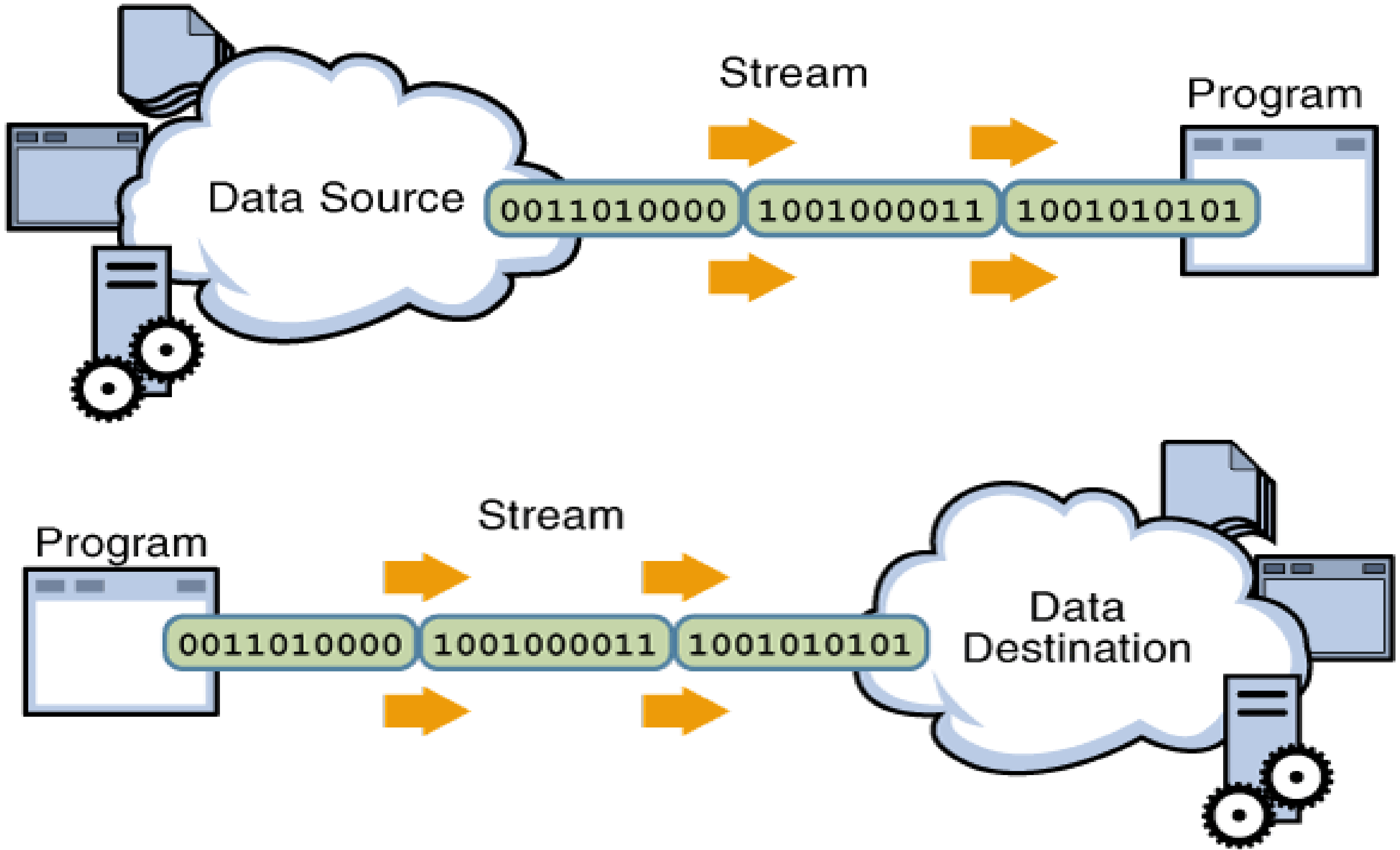
- In Java, streams are simply sequences of bytes

- We can **read** a byte stream from an ***input stream*** object
- We can **write** a byte stream to an ***output stream*** object

- Typically input and output stream objects can be **files** but they also can be **network connections**

- This generality means we can use the same functions for accessing networks as well as reading files

Streams = sequence of data

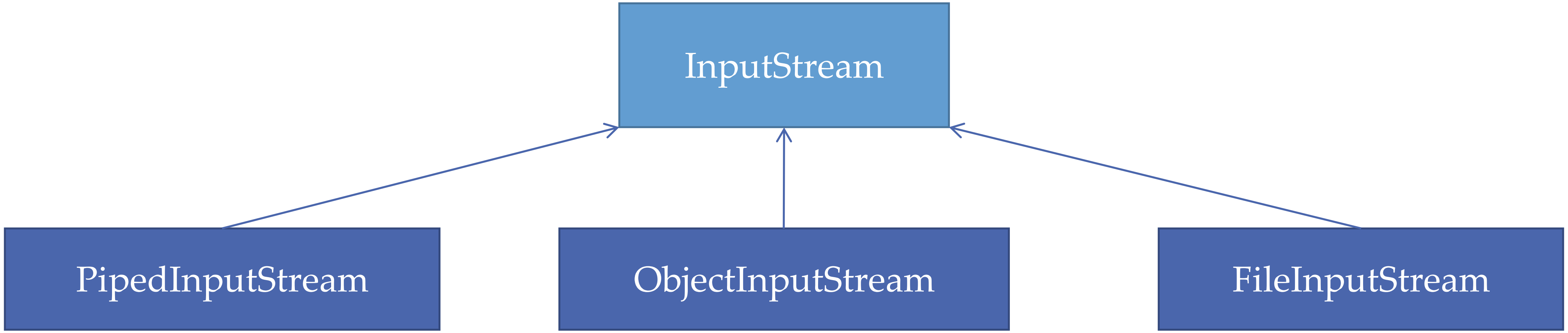


Data Streams

- Streams support many different kinds of data
  - including simple bytes
  - primitive data types
  - localized characters
  - and objects
- Some streams simply pass on data
- Others manipulate and transform the data in useful ways
  - Decorator Design Pattern!

Byte Streams

- All byte stream classes are descended of:
  - InputStream (abstract class)
  - OutputStream (abstract class)
  - Derived must implement the function “int read()”


InputStream

PipedInputStream

ObjectInputStream

FileInputStream

File Streams Example

```
FileInputStream in = null;
FileOutputStream out = null;
out=new FileOutputStream("myFile.dat");
out.write("Hello World!".getBytes());
out.close();

in=new FileInputStream("myFile.dat");
int c;
while((c=in.read()) != -1)
        System.out.print(c+",");

in.close();
```

myFile.dat:
Hello World!

```
104,101,108,108,111,32,87,111,114,108,100,33,
```

Important notes

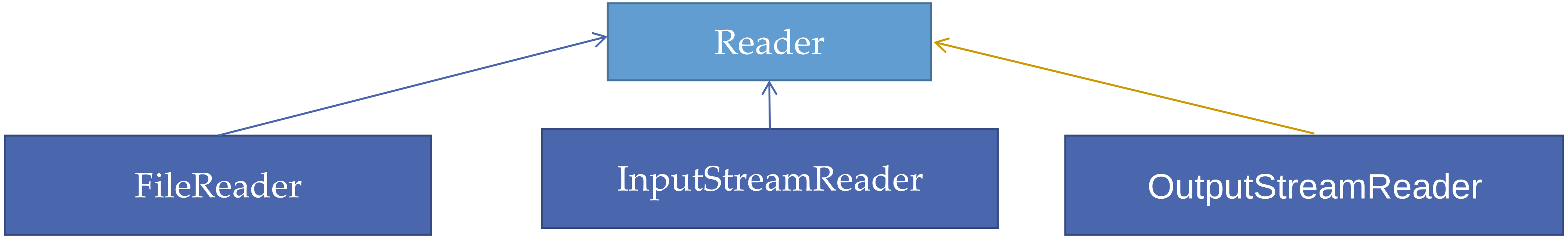
- Always close input/output streams (why?)
- We got the method “close()” from the interface “closeable”

- FileInputStream is very expansive, you should try to avoid using it
  - it’s only good for low level I/O

- Next we will learn efficient ways to read/write characters

Character Streams

- All character stream classes are descended of:
  - Reader (abstract class)
  - Writer (abstract class)
  - Derived must implement the methods
    - read(char[], int, int)
    - close()



File writer / reader Example

```java
FileReader in = null;
FileWriter out = null;
out=new FileWriter("myFile.dat");
out.write("Hello World!");
out.close();


in=new FileReader("myFile.dat");
int c;
while((c=in.read()) != -1)
        System.out.print(c+",");


in.close();
```

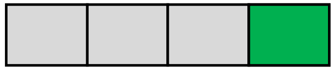
myFile.dat:
Hello World!

```
104,101,108,108,111,32,87,111,114,108,100,33,
```

FileReader v.s FileInputStream

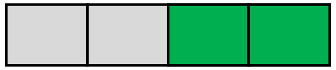
- They both read the file into **int** variable
- FileReader holds the character value in the last **16 bits** of the int
- FileInputStream holds the character value in the last **8 bits** of the int

FileInputStream:
int x=in.read()



$2^8 = 256\,possibilites$

FileReader:
int x=in.read()



$2^{16} = 65536\,possibilites$

Good for Unicode characters!

Line-Oriented I/O

- Sometimes we would like to read an entire line
- A line ends with the characters
  - “\n” or “\r” or “\r\n”
  - depends on the operating system
- We would like to have a mechanism that supports all the variations of “End Line”
- More importantly, we want to save I/O actions

- We need to learn about BufferedStreams

Buffered Streams

- Until now we learned that each read and write request is handled directly by the underlying OS
- Very expensive…

- Java platform implemented buffered I/O Streams
- **Read** to buffer only when buffer is **empty**
- **Write**, only when buffer is **full**

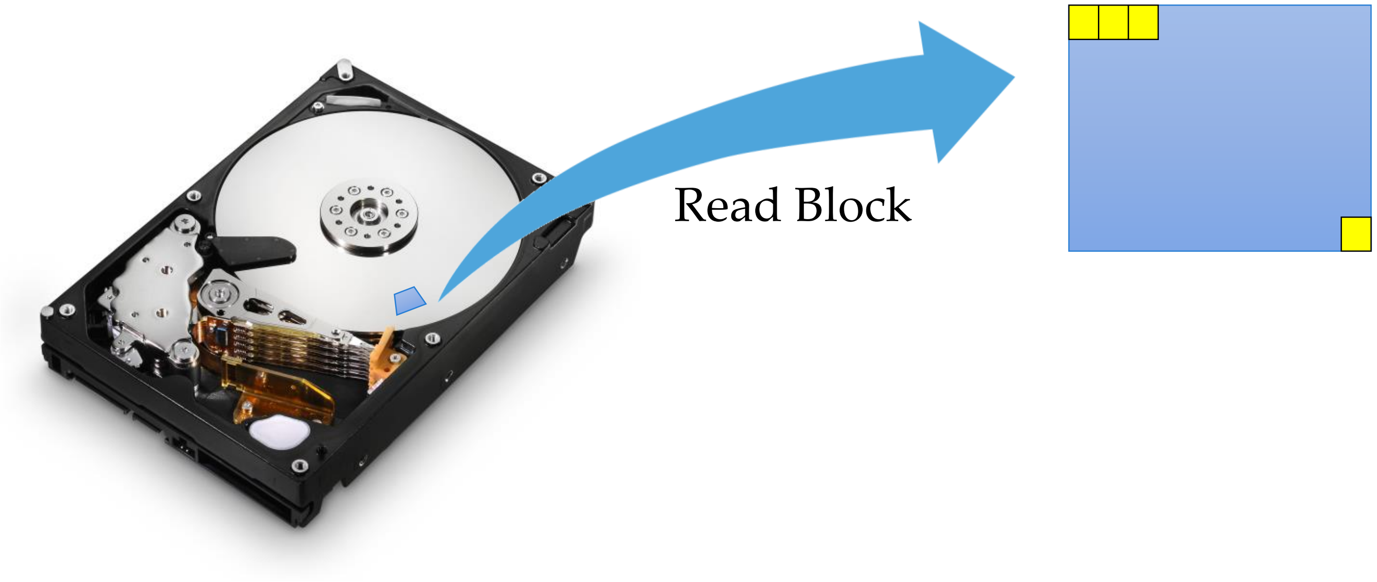
Without a buffer…

I/O source

I/O Action

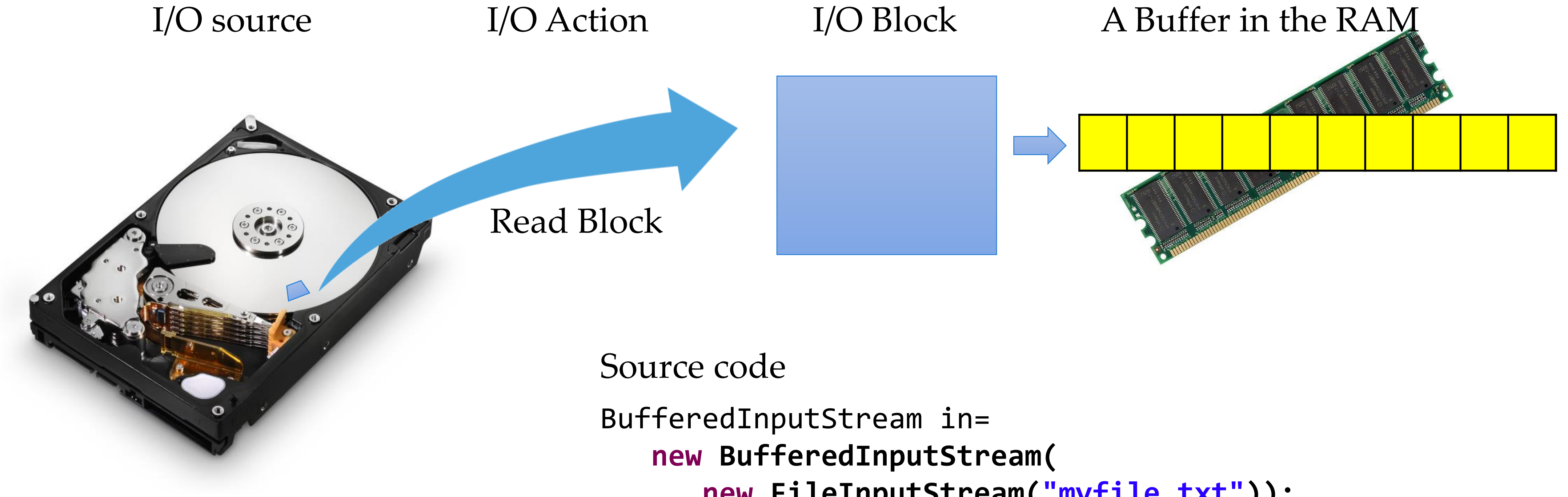
I/O Block

Source code



```
FileInputStream in=
        new FileInputStream("myfile.txt");
int c;
while((c=in.read())!=-1){
    //..
}
```

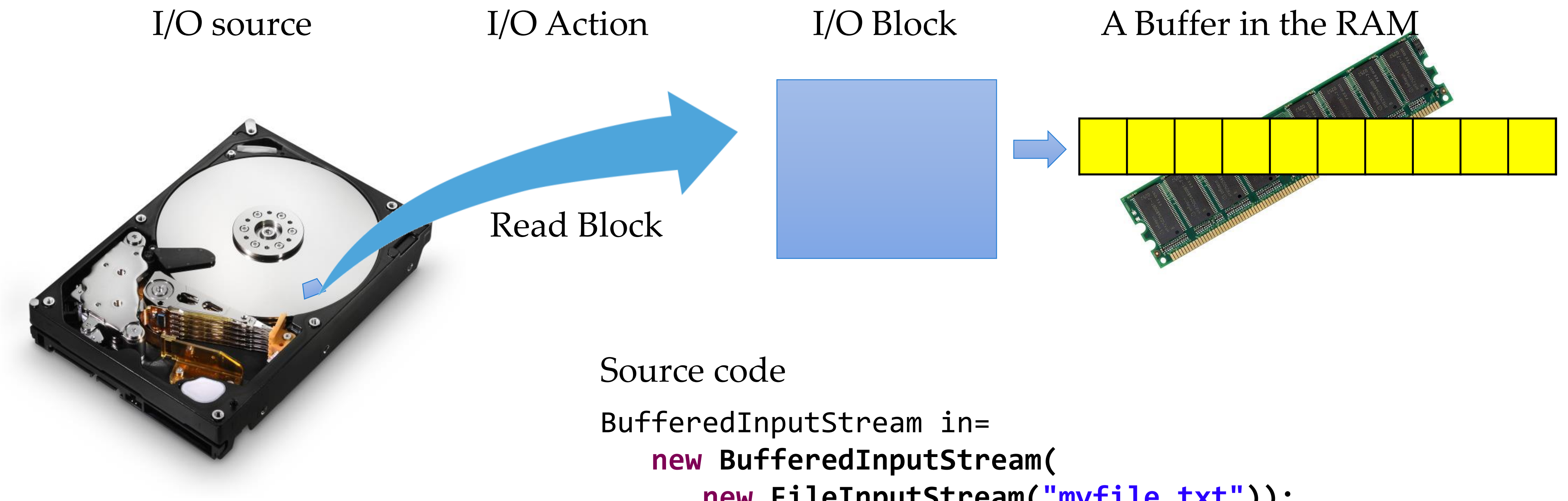
With a buffer…



Source code

```
BufferedInputStream in=
   new BufferedInputStream(
      new FileInputStream("myfile.txt"));
int c;
while((c=in.read())!=-1){
//..
}
```

With a buffer…



I/O source

I/O Action

I/O Block

A Buffer in the RAM

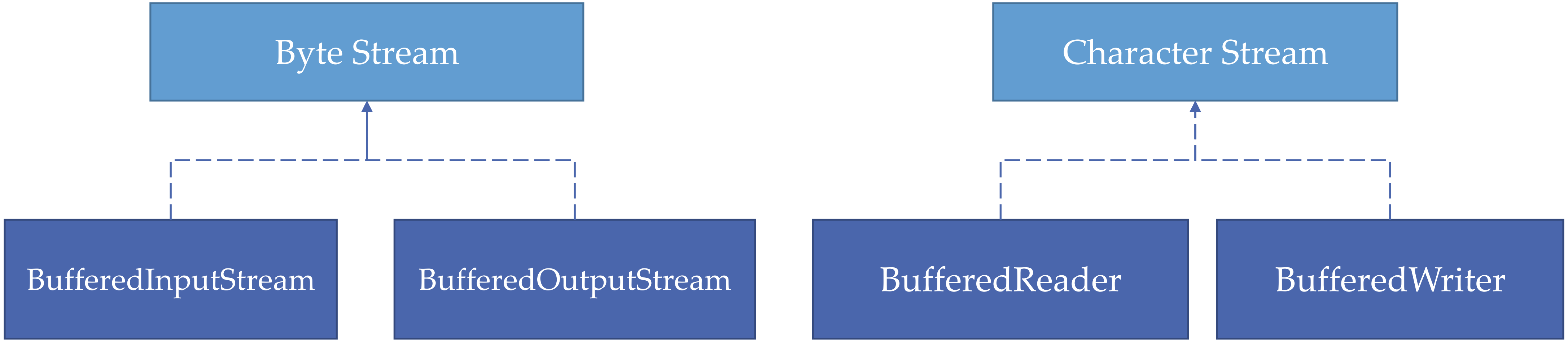
Read Block

Source code

```
BufferedInputStream in=
   new BufferedInputStream(
      new FileInputStream("myfile.txt"));
int c;
while((c=in.read())!=-1){
//..
}
```

Buffered Streams Cont.

- Two kind of Buffers, byte oriented and character oriented.



Buffered Reader/Writer Example

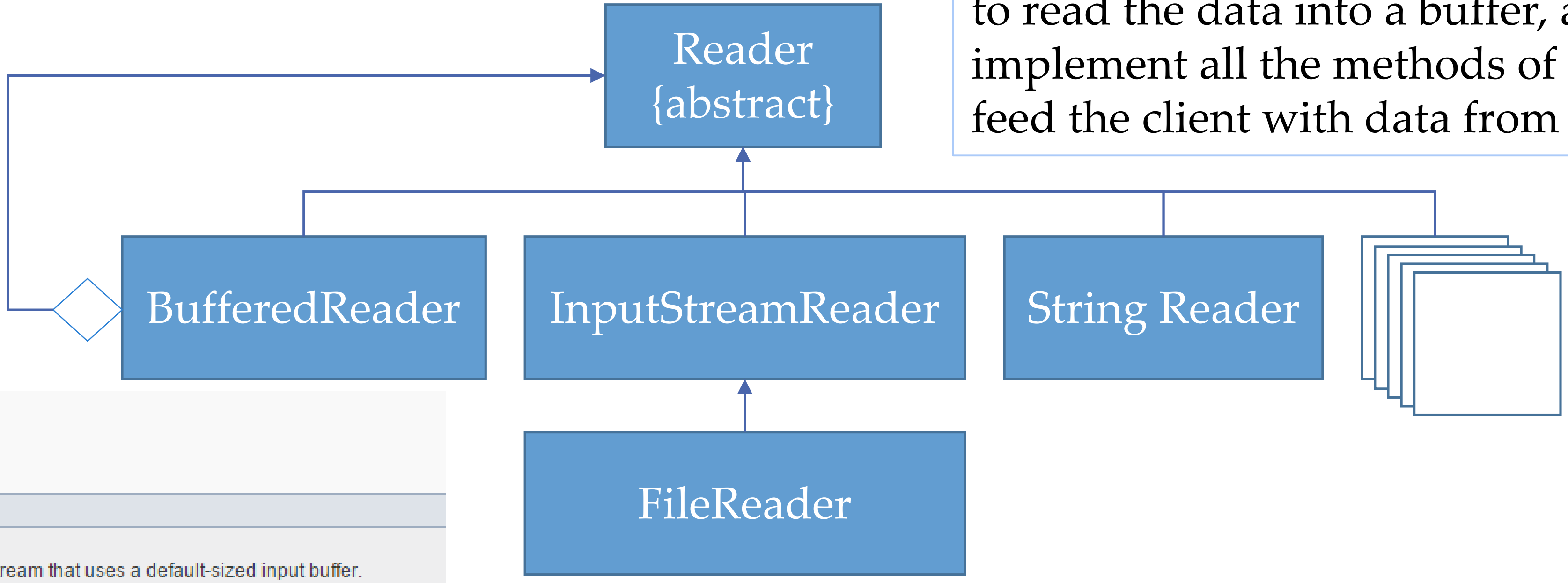
```
BufferedReader reader = null;
PrintWriter writer = null;
reader = new BufferedReader(new FileReader("in.txt"));
writer = new PrintWriter(new FileWriter("out.txt"));

String line;
while ((line = reader.readLine()) != null) {
       writer.println(line);
}

reader.close();
writer.close();
```

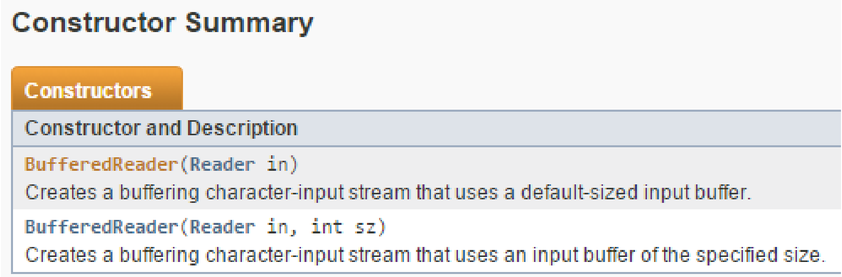
Decorator Design Pattern

```
reader =  new BufferedReader(new FileReader("in.txt"));
```

- What does FileReader do inside the BufferedReader constructor?
- This is a Decorator Design Pattern!

BufferedReader uses its given specific Reader to read the data into a buffer, and implement all the methods of a Reader, and feed the client with data from that buffer





Constructor Summary

Constructors

Constructor and Description
`BufferedReader(Reader in)` Creates a buffering character-input stream that uses a default-sized input buffer.
`BufferedReader(Reader in, int sz)` Creates a buffering character-input stream that uses an input buffer of the specified size.

Flushing Buffered Streams

- Sometimes it makes sense to write out, in a critical point, what is inside the buffer without waiting for it to fill
- We call this action "*flushing*"
- If you want to flush content of the buffer just use the "flush()" method (e.g *bufferName.flush()*)

Scanning

- Java provides an API that breaks input into tokens
  - according to their data type

- Introducing the class “Scanner”

- A simple text scanner which can parse primitive type
  - and strings using regular expressions

Scanning

- A Scanner breaks its input into **tokens**
- Using a **delimiter** pattern
- Which by **default** matches **white-space**
- The resulting tokens may then be converted into values of different types
- Using the various **next** methods

- It is like an iterator!!!

Scanner example

```
Scanner myScaner=null;
myScaner=new Scanner(
                 new BufferedReader(
                      new FileReader("in.txt")));

while(myScaner.hasNext()){
 System.out.println(myScaner.next());
}
```

Scanner example

```
String input = "8.5 32,767 3.14159 1,000,000.1";
Scanner s = new Scanner(input);
double sum=0;
while(s.hasNextDouble())
     sum+=s.nextDouble();

s.close();
System.out.println(sum);
```

```
1032778.74159
```

Change Delimiters

- To use different token separator use:
- useDelimiter(String pattern)
- pattern – a string specifying a delimiting pattern
- For example:

```
String input="1 fish 2 fish red fish blue";
Scanner s=new Scanner(input);
s.useDelimiter(" fish ");
System.out.println(s.nextInt());
System.out.println(s.nextInt());
System.out.println(s.next());
System.out.println(s.next());
```

Output:

```
1
2
red
blue
```

I/O from command

- A program is often run from the command line
- and interacts with the user in the command line environment

- The Java platform supports this kind of interaction
- Standard Streams are a feature of many operating systems
- By default,
  - they read input from the **keyboard**
  - and write output to the **display**

Standard streams

- Standard input – *System.**in***
- Standard output – System.***out***
- Standard error – System.***err***
- These objects are defined automatically and do not need to be opened
- Standard Streams are byte streams for historical reasons
- But we like character streams, what can we do with System.in?

Wrapping System.in

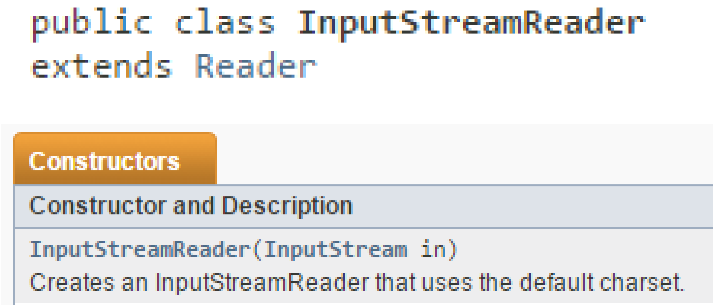
- We can wrap *System.in* with *InputStreamReader* and even *BufferedReader*:

```
BufferedReader in = new BufferedReader(
     new InputStreamReader(System.in));

String line = in.readLine();
```

What is this design pattern??

It is an Object Adapter Pattern!

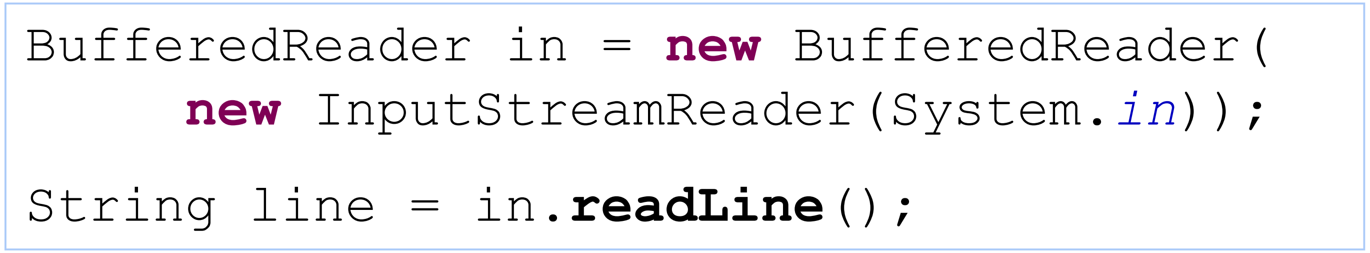
Javadoc:



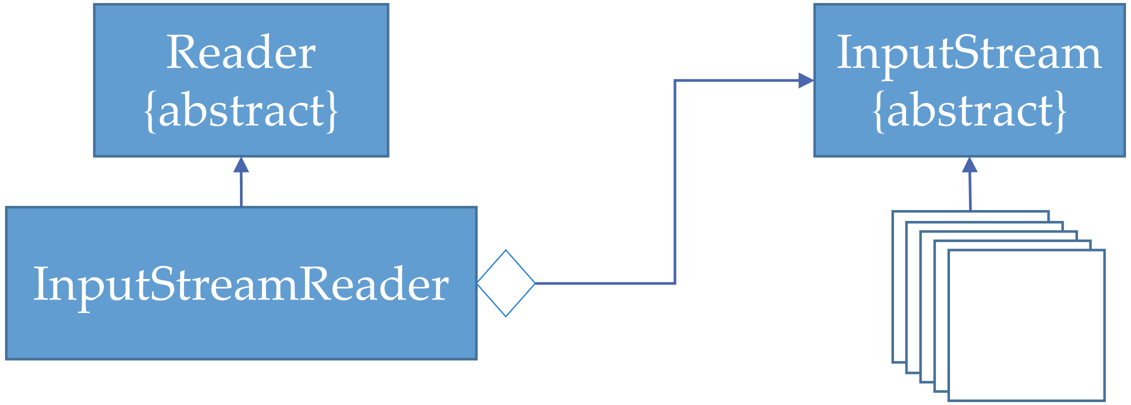
```
public class InputStreamReader
extends Reader
```

Constructors

Constructor and Description
`InputStreamReader(InputStream in)` Creates an InputStreamReader that uses the default charset.



```
BufferedReader in = new BufferedReader(
     new InputStreamReader(System.in));

String line = in.readLine();
```



Object Streams

- Object streams support I/O of objects
- Most, but not all, standard classes support serialization of their objects
- Those that do, implement the Serializable interface
  - Have a default CTOR
  - Have setters and getters to all data members
- The object stream classes are:
  - ObjectInputStream and ObjectOutputStream
- These classes implement
  - ObjectInput and ObjectOutput

Object Streams

- When you write your object **all** of it’s sub-objects
- must implement Serializable interface

- When you read a Serialized object from file
- you must do casting to the returned value
- because the returned value is “Object” type

Object Streams example

```
public class Point implements Serializable{
 int x,y;
 public Point(int x,int y){
  this.x=x;
  this.y=y;
}
 public String toString(){
  return "("+x+","+y+")";
 }
 // don’t forget default CTOR, setters & getters
}
```

Object Streams example

```
public class Line implements Serializable{
 Point p1,p2;
 public Line(Point p1,Point p2){
  this.p1=p1;
  this.p2=p2;
 }
 public String toString(){
  return "p1= "+p1+" p2= "+p2;
 }
 // don’t forget default CTOR, setters & getters
}
```

Object Streams example

```
Line a=new Line(new Point(2, 3), new Point(4, 5));

System.out.println(a);

ObjectOutputStream out= new ObjectOutputStream(
                         new FileOutputStream("out.txt"));

out.writeObject(a);


ObjectInputStream in= new ObjectInputStream(
                      new FileInputStream("out.txt"));

Line b=(Line) in.readObject();

System.out.println(b);
```