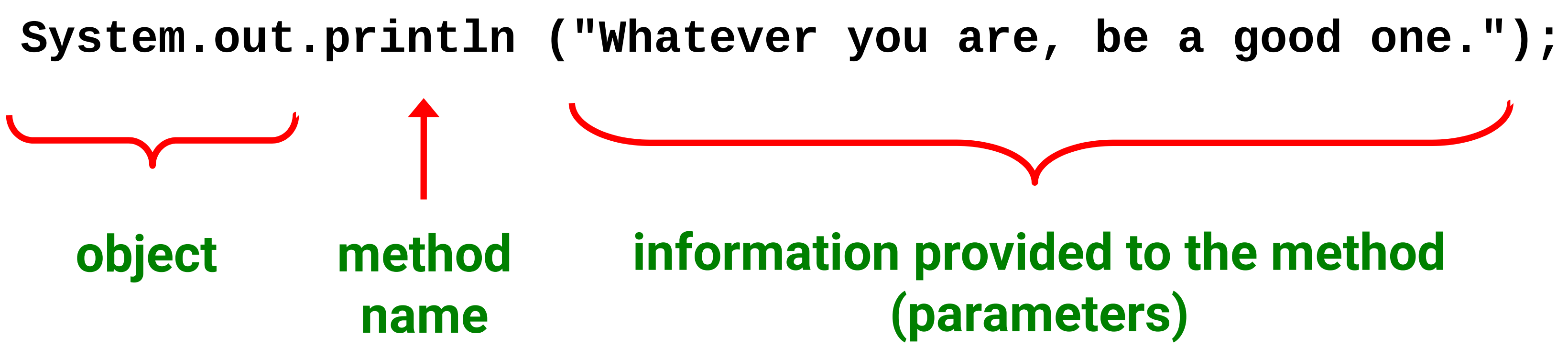
Object Oriented Programming with Java

Input/Output & Streams

The println Method

- The **`System.out`** object represents a destination (the monitor screen) to which we can send text output



```
System.out.println ("Whatever you are, be a good one.");
```

The print Method

- The **`System.out`** object also provides the print method. The **`print`** method is like the **`println`** method, except that it does not advance to the next line
- Therefore anything printed after a **`print`** statement will appear on the same line

```java
System.out.print("line one. ");
System.out.print("also line one. ");
```

Interactive Programs

- Programs need input on which to operate
- The `Scanner` class aids reading values of various types
- A `Scanner` object can be set up to read input from various sources, including the keyboard or even a specific String.
- Keyboard input is represented by the `System.in` object

Reading Input

- The following line creates a **Scanner** object that reads from the keyboard:

```
Scanner a = new Scanner(System.in);
```

- The **new** operator creates the **Scanner** object
- Once created, the **Scanner** object can be used to invoke various input methods, such as:

```
a.nextLine();
```

Reading Input

- The **`Scanner`** class is part of the **`java.util`** class library, and must be imported into a program to be used. (add import java.util.Scanner; at top of file)
- The **`nextLine`** method reads all of the input until the end of the line is found
- The details of object creation and class libraries are discussed further in Chapter 3

The System class

- Refers to the operating system, which handles input/output for programs you write
  - `System.out`
  - `System.in`
  - `System.err`
- These are all *buffers* you have access to from the System class

Input Tokens

- Unless specified otherwise, *white space* is used to separate the elements (called *tokens*) of the input
- White space includes space characters, tabs, new line characters
- The **`next`** method of the **`Scanner`** class reads the next input token and returns it as a string
- Methods such as **`nextInt`** and **`nextDouble`** read data of particular types (they convert the next token to int or double)

Processing Input

From the Java **Scanner**. What does this do?

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```

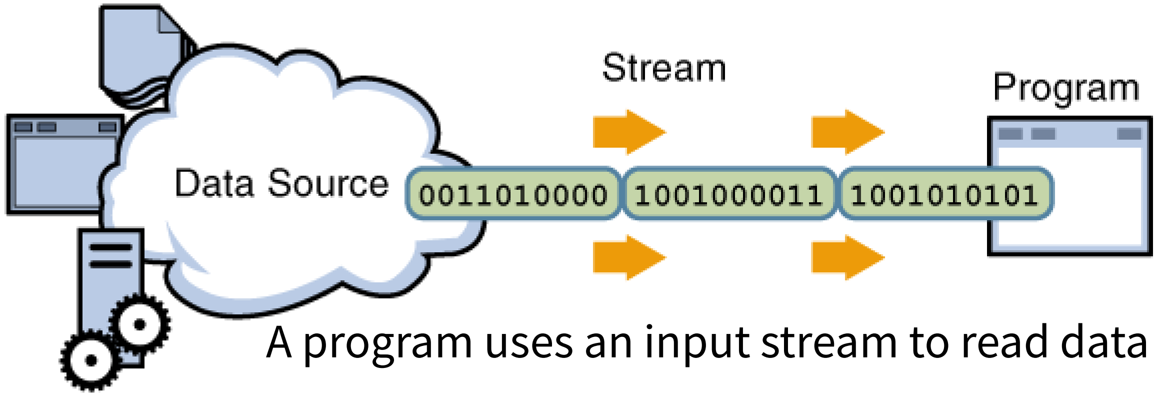
How about this?

```
String s = sc.nextLine();
```

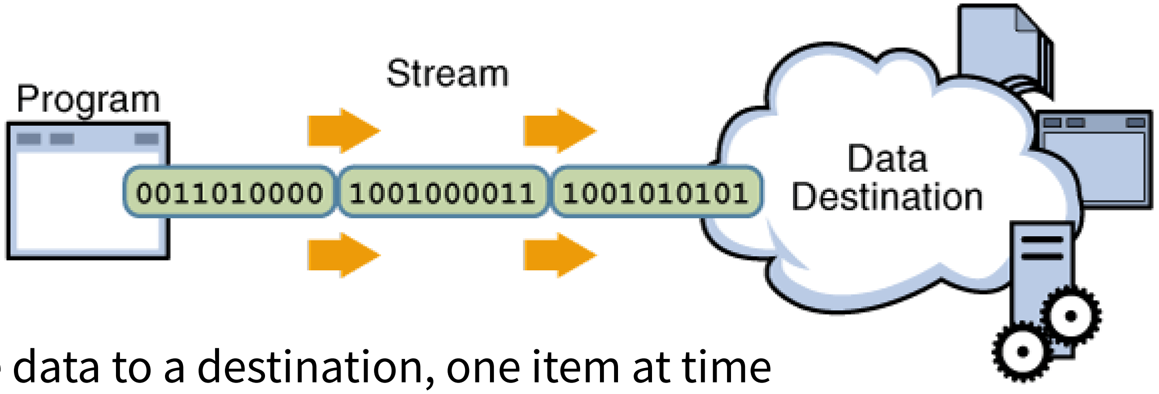
Note: we only need to create the Scanner once to call its nextFoo methods multiple times, in the order we choose.

Input/Output as a Stream

Java views input/output as a stream of bytes regardless of the source/destination (file, network, screen, keyboard, printer, etc.)



A program uses an input stream to read data from a source, one item at a time



A program uses an output stream to write data to a destination, one item at time

Tutorial: https://docs.oracle.com/javase/tutorial/essential/io/streams.html

I/O Classes Hierarchy – java.io

```
java.lang.Object
    java.io.File

    java.io.InputStream                         abstract class
        java.io.ByteArrayInputStream
        java.io.FileInputStream                 low-level Byte Stream; binary files; avoid in text files w/ encoding
        java.io.FilterInputStream
            java.io.BufferedInputStream
            java.io.DataInputStream
            java.io.LineNumberInputStream
            java.io.PushbackInputStream
        java.io.ObjectInputStream
        java.io.PipedInputStream
        java.io.SequenceInputStream
        java.io.StringBufferInputStream
    java.io.ObjectInputStream.GetField
    java.io.ObjectOutputStream.PutField
    java.io.ObjectStreamClass
    java.io.ObjectStreamField

    java.io.OutputStream                        abstract class
        java.io.ByteArrayOutputStream
        java.io.FileOutputStream                low-level Byte Stream; binary files; avoid in text files w/ encoding
        java.io.FilterOutputStream
            java.io.BufferedOutputStream
            java.io.DataOutputStream
            java.io.PrintStream
        java.io.ObjectOutputStream
        java.io.PipedOutputStream
```

I/O Classes Hierarchy – java.io

```
java.lang.Object
    java.io.RandomAccessFile

    java.io.Reader                              abstract class
        java.io.BufferedReader
            java.io.LineNumberReader
        java.io.CharArrayReader
        java.io.FilterReader
            java.io.PushbackReader
        java.io.InputStreamReader               for text files in non-default host's character encoding
            java.io.FileReader                  Character Stream; text files; uses FileInputStream for the physical I/O
        java.io.PipedReader
        java.io.StringReader
    java.io.StreamTokenizer

    java.io.Writer                              abstract class
        java.io.BufferedWriter
        java.io.CharArrayWriter
        java.io.FilterWriter
        java.io.OutputStreamWriter
            java.io.FileWriter                  Character Stream; text files; uses FileOutputStream for the physical I/O
        java.io.PipedWriter
        java.io.PrintWriter
        java.io.StringWriter
```

not an exhaustive list of java.io

Non-blocking I/O Classes Hierarchy – java.nio

```
java.lang.Object
    java.nio.Buffer
        java.nio.ByteBuffer
            java.nio.MappedByteBuffer
        java.nio.CharBuffer
        java.nio.DoubleBuffer
        java.nio.FloatBuffer
        java.nio.IntBuffer
        java.nio.LongBuffer
        java.nio.ShortBuffer
    java.nio.ByteOrder
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.nio.BufferOverflowException
                java.nio.BufferUnderflowException
                java.lang.IllegalStateException
                    java.nio.InvalidMarkException
                java.lang.UnsupportedOperationException
                    java.nio.ReadOnlyBufferException
```

I/O Classes Hierarchy... more

As if that wasn't enough, we also have **`java.util.Scanner`** that we've been using so far for basic input

- BufferedReader is synchronous while Scanner is not. BufferedReader should be used if we are working with multiple threads.
- BufferedReader has significantly larger buffer memory than Scanner.
- The Scanner has a little buffer (1KB char buffer) as opposed to the BufferedReader (8KB byte buffer), but it's more than enough most of the time.
- BufferedReader is a bit faster as compared to Scanner because scanner does parsing of input data and BufferedReader simply reads sequence of characters.

...and **`java.util.Formatter`** for basic (C-like) formatted output

Working with Files

- We can get input from files just as easily as from the keyboard, using a **Scanner**
- We can write to a file as easily as to the terminal, using a **PrintWriter**
- The file extension is arbitrary (`.txt`, `.csv`, .etc). The file just contains a sequence of characters that we can use however we choose.
- The One Hitch: Java requires us to **`try-catch`** any exceptions such as **`FileNotFoundException`**

Simple reading from Files

We can get input from files just as easily as from the keyboard

```
import java.util.Scanner;  //outside the class

try
{
      Scanner sc = new Scanner (new File("outs.txt"));
      String s = "";
      while (sc.hasNextLine())
      {
        s += sc.nextLine()+"\n";
      }
      System.out.print("contents: \n"+s);
}
catch (FileNotFoundException e)
{
      System.out.println("file not present... :( ");
}
```

Simple writing to Files

We can write strings to files just as easily as the terminal

```
import java.io.PrintWriter;  // outside the class

try
{
      PrintWriter pw = new PrintWriter(new File("outs.txt"));
      pw.print("writing a file from a program! :) \na\nb\nc");
      pw.close();
}
catch (FileNotFoundException e)
{
      System.out.println("file not found…  >:|");
}
```

Practice Problems

- Use a **PrintWriter** to write the numbers 1-100 to a file.
- Use a **Scanner** attached to that file to read in the numbers into an array; find the sum of them.
- Write a program that asks for a number, then calculates all the primes less than that number, writing them to **`primes_under_n.txt`**
  (where n is the number they gave you)

XML serialization example

```java
import java.io.BufferedWriter;
import java.nio.file.Files;
import java.nio.file.Paths;
import javax.xml.bind.JAXB;

public class CreateXML {
    public static void main(String[] args) {
    BufferedWriter out = Files.newBufferedWriter(Paths.get("file.xml"));
    Obj = new...
    JAXB.marshal(obj, out);
    }
}
```