Chapter 9 - Input / Output

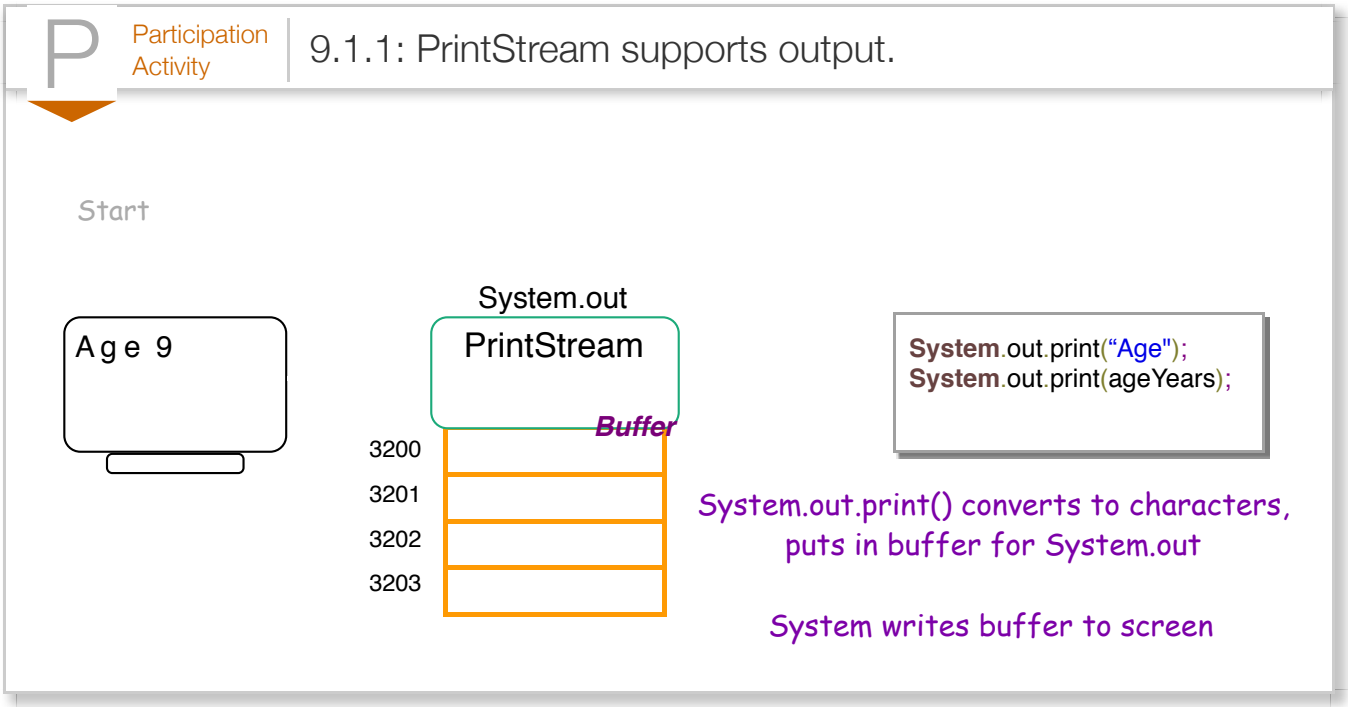
Section 9.1 - OutputStream and System.out

Programs need a way to output data to a screen, file, or elsewhere. An ***OutputStream*** is a class that supports output. OutputStream provides several overloaded methods for writing a sequence of bytes to a destination. That sequence is normally placed into a buffer, and the system then outputs the buffer at various times.

System.out is a predefined output stream object reference that is associated with a system's standard output, usually a computer screen. The System class' out variable is a reference derived from OutputStream called a ***PrintStream*** (e.g., `PrintStream out;` in the System class). The PrintStream class extends the base functionality of the OutputStream class and provides the print() and println() methods for converting different types of data into a sequence of characters. The following animation illustrates.

Participation Activity

9.1.1: PrintStream supports output.



The print() and println() methods are overloaded in order to support the various standard data types, such as int, boolean, float, etc., each method converting that data type to a sequence of characters. Basic use of these methods and System.out was discussed in an earlier section.

The print() and println() methods also provide support for reference types. When a programmer invokes either printing method with an argument of a reference type, the method prints a string representation of the object. This string representation consists of the name of the object's class followed by the "@" character and the hexadecimal value of the object's hash code. A hash code typically represents the object's address in memory, although this is not guaranteed by the Java specification.

Note that because the System class is predefined, a programmer is not required to import the System class in order to use the output stream System.out.

Participation Activity

9.1.2: ostream and System.out.

#	Question	Your answer
1	Characters written to System.out are immediately written to a system's standard output.	True
		False
2	To use System.out, a program must include the statement import `java.io.PrintStream;`.	True
		False
3	Various standard data types are converted to a character sequence by print() and println().	True
		False
4	The output of print() and println() for a reference type includes the object's class.	True
		False

Exploring further:

- Oracle's OutputStream class specification
- Oracle's PrintStream class specification
- Oracle's System class specification

Section 9.2 - InputStream and System.in

Programs need a way to receive input data, from a keyboard, touchscreen, or elsewhere. An ***InputStream*** is a class for achieving such input. InputStream provides several overloaded read() methods that allow a programmer to extract bytes from a particular source.

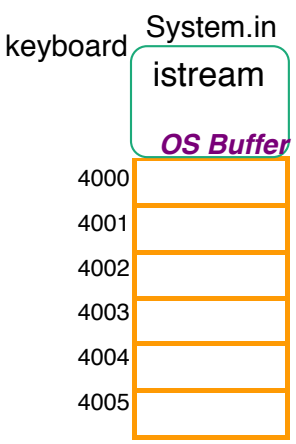
System.in is a predefined input stream object reference that is associated with a system's standard input, which is usually a keyboard. A programmer is not required to import the System class in order to use System.in because the System class is a predefined class.

The System.in input stream automatically reads the standard input from a memory region, known as a buffer, that the operating system fills with the input data. The following animation illustrates.

Participation Activity

9.2.1: InputStream supports byte input.

Start

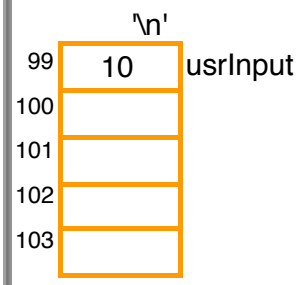



```
import java.io.IOException;

public class InputStreamReader {
   public static void main (String[] args)
   throws IOException {
      int usrInput = 0;

    // Read 1st byte
      usrInput = System.in.read();
    // Read 2nd byte
      usrInput = System.in.read();
    // Read 3rd byte
      usrInput = System.in.read();
    // Read 4th byte
      usrInput = System.in.read();
    // Read 5th byte
      usrInput = System.in.read();
    // Read 6th byte
      usrInput = System.in.read();

      return;
   }
}
```



Operating System puts input characters in buffer

Participation Activity

9.2.2: Input streams and System.in.

#	Question	Your answer
1	System.in is a predefined InputStream associated with the system's standard input.	True
		False
2	A program must import the InputStream class in order to use System.in.	True
		False
3	A read from System.in will read bytes from a buffer filled by the operating system.	True
		False

System.in is an input ***byte stream***, and thus the read() method reads the first 8-bit ASCII value available from the operating system's buffer. Each 8-bit value read from the input stream is returned as an int (instead of byte) in order to allow the programmer to determine if data is no longer available, which is indicated by a return value of -1.

When using an InputStream, a programmer must append the clause `throws IOException` to the definition of main(), as seen in the above animation. A ***throws clause*** tells the Java virtual machine that the corresponding method may exit unexpectedly due to an ***exception***, which is an event that disrupts a program's execution. In this case, the throws clause indicates that the program may terminate due to an input/output exception (i.e., IOException). A program can throw an IOException when it encounters certain errors as it is trying to read from an input stream. Exceptions and how to handle them are discussed in more detail elsewhere. For now, it is sufficient to know that certain input and output streams require the programmer to append a throws clause to the definition of main().

As seen from previous examples, a programmer often needs a way to extract strings or integers from an input stream. Instead of directly reading bytes from System.in, a program typically uses the Scanner class as a wrapper that augments System.in by automatically scanning a sequence of bytes and converting those bytes to the desired data type. To initialize a Scanner object, a programmer can

pass an InputStream, such as System.in, as an argument to the constructor as in `Scanner scnr = new Scanner(System.in);`. Basic use of the Scanner object is discussed in earlier sections.

Participation Activity

9.2.3: Byte stream and throws clause.

#	Question	Your answer
1	How many bits of data are returned by System.in.read()?	
2	What value is returned by System.in.read() when data is no longer available?	
3	What clause needs to be appended to the definition of main() when using an InputStream?	

Exploring further:

- Oracle's Java tutorials on I/O Streams
- Oracle's InputStream class specification

Section 9.3 - Output formatting

A programmer can adjust the way that output appears, a task known as ***output formatting***. The standard output stream System.out provides the methods ***printf()*** and ***format()*** for this task. Both methods are equivalent, so this discussion will only refer to printf().

Like the print() and println() methods, printf() allows a programmer to print text along with variable

values. printf(), however, affords the programmer more freedom in specifying the format of the output.

The first argument of the printf() method is referred to as the ***format string***, which defines the format of the text that will be printed along with any number of placeholders for printing numeric values. These placeholders are known as format specifiers. ***Format specifiers*** define the type of values being printed in place of the format specifier.

A format specifier begins with the % character followed by a sequence of characters that indicate the type of value to be printed. For each format specifier within the format string, the value to be printed must be provided in the printf() statement as arguments following the format strings. These arguments are additional input to the printf() method, with each argument separated by a comma within the parentheses. The following is an example of a printf() statement that prints a sentence including a single decimal integer value.

Figure 9.3.1: Single decimal printf statement example.

```
System.out.printf(" You know %d people.\n", totalPpl);
```

The %d format specifier in the example above indicates that the printf() statement should output a decimal integer value. Specifically, the % indicates we would like to output a value stored within a variable, and the d indicates how we would like that value to be displayed as a decimal integer. Following the format string (separated by a comma), the variable total indicates that the value stored within this variable will be printed in place of the %d format specifier.

All format specifiers begin with %, thus % is a special character. To print a % character using printf(), the sequence %% is used, as in:
`printf("Annual percentage rate is %f %%.\n", rate);`.

Multiple format specifiers can appear within the format string. The %f is used for printing floating-point values, such as float and double. The value stored within the variable years will be printed as a decimal integer in place of the %d format specifier, and the value stored within the variable total will be displayed as a double.

Figure 9.3.2: Multiple format specifiers within a format string.

```
System.out.printf("Savings after %d years is: %f\n\n", years, total);
```

In addition to numeric values, the format specifiers can also be used to print individual characters (using the format specifier %c) and strings (using the format specifier %s). The following table provides an overview of the format specifiers required for various data types.

Table 9.3.1: Format specifiers for the printf() and format() methods.

Format specifier	Data Type(s)	Notes
%c	char	Prints a single Unicode character
%d	int, long, short	Prints a decimal integer value.
%o	int, long, short	Prints an octal integer value.
%h	int, char, long, short	Prints a hexadecimal integer value.
%f	float, double	Prints a floating-point value.
%e	float, double	Prints a floating-point value in scientific notation.
%s	String	Prints the characters in a String variable or literal.
%%		Prints the '%' character.
%n		Prints the platform-specific new-line character.

The format specifiers within the format string of printf() can include ***format sub-specifiers***. These sub-specifiers define how a value stored within a variable will be printed in place of a format specifier.

The formatting sub-specifiers are included between the % and format specifier characters. For example, `printf("%.1f", myFloat);` causes the floating-point variable, myFloat, to be output with only 1 digit after the decimal point; if myFloat was 12.34, the output would be 12.3. Format specifiers and sub-specifiers use the following form:

Construct 9.3.1: Format specifiers and sub-specifiers.

```
%(flags)(width)(.precision)specifier
```

Floating point values

Formatting floating-point output is commonly done using the following sub-specifiers options. For the following assume myFloat has a value of 12.34. Recall that %f is used for floating-point values and %e is used to display floating-point values in scientific notation.

Table 9.3.2: Floating-point formatting.

Method calls to printf() apply to PrintStream objects like System.out.

Sub-specifier	Description	Example
width	Specifies the minimum number of characters to be printed. If the formatted value has more characters than the width, it will not be truncated. If the formatted value has fewer characters than the width, the output will be padded with spaces (or 0's if the '0' flag is specified).	`printf("Value: %7.2f", myFloat);` `Value:   12.34`

.precision	Specifies the number of digits to print following the decimal point. If the precision is not specified a default precision of 6 is used.	`printf("%.4f", myFloat);`
		`12.3400`
		
		`printf("%3.4e", myFloat);`
		`1.2340e+01`

flags	-: Left justifies the output given the specified width, padding the output with spaces.
	+: Print a preceding + sign for positive values. Negative numbers are always printed with the - sign.
	0: Pads the output with 0's when the formatted value has fewer characters than the width.
	space: Prints a preceding space for positive value.
		`printf("%+f", myFloat);`
		`+12.340000`
		`printf("%08.2f", myFloat);`
		`00012.34`

Figure 9.3.3: Example output formatting for floating-point numbers.

```
import java.util.Scanner;

public class FlyDrive {
   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      double miles = 0.0;     // User defined distance
      double hrsFly = 0.0;    // Time to fly distance
      double hrsDrive = 0.0; // Time to drive distance

      // Prompt user for distance
      System.out.print("Enter a distance in miles: ");
      miles = scnr.nextDouble();
      // Calculate the correspond time to fly/drive dis

      hrsFly = miles / 500.0;
      hrsDrive = miles / 60.0;

      // Output resulting values
      System.out.printf("%.2f miles would take:\n", miles);
      System.out.printf("%.2f hours to fly\n", hrsFly);
      System.out.printf("%.2f hours to drive\n", hrsDrive);

      return;
   }
}
```

```
Enter a distance in miles: 1
10.30 miles would take:
0.02 hours to fly
0.17 hours to drive
```

Participation Activity

9.3.1: Formatting floating point outputs using printf().

What is the output from the following print statements, assuming

```
float myFloat = 45.1342;
```

#	Question	Your answer
1	`printf("%09.3f", myFloat);`	
2	`printf("%.3e", myFloat);`	
3	`printf("%09.2f", myFloat);`	

Integer values

Formatting of integer values can also be done using sub-specifiers. The behavior of sub-specifiers for integer data behave differently than for floating-point values. For the following assume myInt is an int value of 301.

Table 9.3.3: Integer formatting.

Method calls to printf() apply to PrintStream objects like System.out.

Sub-specifier	Description	Example
width	Specifies the minimum number of characters to be printed. If the formatted value has more characters than the width, it will not be truncated. If the formatted value has fewer characters than the width, the output will be padded with spaces (or 0's if the '0' flag is specified).	`printf("Value: %7d", myInt);` `Value:     301`

flags	-: Left justifies the output given the specified width, padding the output with spaces.
	+: Print a preceding + sign for positive values. Negative numbers are always printed with the - sign.
	0: Pads the output with 0's when the formatted value has fewer characters than the width.
	space: Prints a preceding space for positive value.
		`printf("%+d", myInt);`
		`+301`
		
		`printf("%08d", myInt);`
		`00000301`
		
		`printf("%+08d", myInt);`
		`+0000301`

Figure 9.3.4: Output formatting for integers.

```
public class CelestialBodyDist {
   public static void main(String[] args) {
      final long KM_EARTH_TO_SUN = 149598000;    // Dist from Earth to sun
      final long KM_SATURN_TO_SUN = 1433449370;  // Dist from Saturn to sun

      // Output distances with min number of characters
      System.out.printf("Earth is %12d", KM_EARTH_TO_SUN);
      System.out.printf(" kilometers from the sun.\n");
      System.out.printf("Saturn is %11d", KM_SATURN_TO_SUN);
      System.out.printf(" kilometers from the sun.\n");

      return;
   }
}
```

```
Earth is    149598000 kilometers from the sun.
Saturn is  1433449370 kilometers from the sun.
```

Participation Activity

9.3.2: Formatting integer outputs using printf().

What is the output from the following print statements, assuming

```
int myInt = -713;
```

#	Question	Your answer
1	`printf("%+04d", myInt);`	
2	`printf("%05d", myInt);`	
3	`printf("%+02d", myInt);`	

Strings

Formatting of strings can also be done using sub-specifiers. For the following assume myString is the string "Formatting".

Table 9.3.4: String formatting.

Method calls to printf() apply to PrintStream objects like System.out.

Sub-specifier	Description	Example
width	Specifies the minimum number of characters to be printed. If the string has more characters than the width, it will not be truncated. If the formatted value has fewer characters than the width, the output will be padded with spaces.	`printf("%20s String", myString);` `Formatting String`
.precision	Specifies the maximum number of characters to be printed. If the string has more characters than the precision, it will be truncated.	`printf("%.6s", myString);` `Format`
flags	-: Left justifies the output given the specified width, padding the output with spaces.	`printf("%-20s String", myString);` `Formatting String`

Figure 9.3.5: Example output formatting for Strings.

```
public class DogAge {
   public static void main(String[] args) {

      System.out.printf("Dog age in human years (dogyears.com)\n\n");
      System.out.printf("-------------------------\n");

         // set num char for each column, left justified
      System.out.printf("%-10s | %-12s\n", "Dog age", "Human age");
      System.out.printf("-------------------------\n");

      // set num char for each column, first col left justified
      System.out.printf("%-10s | %12s\n", "2 months", "14 months");
      System.out.printf("%-10s | %12s\n", "6 months", "5 years");
      System.out.printf("%-10s | %12s\n", "8 months", "9 years");
      System.out.printf("%-10s | %12s\n", "1 year", "15 years");
      System.out.printf("-------------------------\n");

      return;
   }
}
```

```
Dog age in human ye

--------------------
Dog age    | Human
--------------------
2 months   |    14
6 months   |     5
8 months   |     9
1 year     |    15
--------------------
```

Participation Activity

9.3.3: Formatting string outputs using printf().

What is the output from the following print statements, assuming

```
String myString = "Testing";
```

Make sure all of your responses are in qutoes, e.g. "Test".

#	Question	Your answer
1	`printf("%4s", myString);`	
2	`printf("%8s", myString);`	
3	`printf("%.4s", myString);`	
4	`printf("%.10s", myString);`	

Flushing output

Printing characters from the buffer to the output device (e.g., screen) requires a time-consuming reservation of processor resources; once those resources are reserved, moving characters is fast, whether there is 1 character or 50 characters to print. As such, the system may wait until the buffer is full, or at least has a certain number of characters before moving them to the output device. Or, with fewer characters in the buffer, the system may wait until the resources are not busy. However, sometimes a programmer does not want the system to wait. For example, in a very processor-intensive program, such waiting could cause delayed and/or jittery output. The programmer can use a PrintStream's (e.g., System.out) method ***flush()***. The flush() method will immediately flush the contents of the buffer for the specified OutputStream. For example, the statement System.out.flush() will write the contents of the buffer for System.out to the computer screen.

Exploring further:

- More formatting options exist. See Oracle's Java Formatter class specification.

Challenge Activity

9.3.1: Output formatting: Printing a maximum number of decimals.

Write a single statement that prints outsideTemperature with 2 decimals. End with newline. Sample o

```
103.46
```

```
1 public class OutsideTemperatureFormatting {
2    public static void main (String [] args) {
3       double outsideTemperature = 103.46432;
4
5       /* Your solution goes here  */
6
7       return;
8    }
9 }
```

Run

Section 9.4 - Streams using Strings

Sometimes a programmer wishes to read input data from a string rather than from the keyboard (standard input). A programmer can associate a Scanner object with a String rather than with the keyboard (standard input). Such an object can be used just like a Scanner object associated with the System.in stream. A Scanner object initialized from a String is often referred to as an ***input string stream***. The following program illustrates.

Figure 9.4.1: Reading from a String using a Scanner object.

```
import java.util.Scanner;

public class StringInputStream {
   public static void main(String[] args) {
      Scanner inSS = null;               // Input string stream
      String userInfo = "Amy Smith 19"; // Input string
      String firstName = "";             // First name
      String lastName =  "";             // Last name
      int userAge = 0;                   // Age

      // Init scanner object with string
      inSS = new Scanner(userInfo);

      // Parse name and age values from string
      firstName = inSS.next();
      lastName = inSS.next();
      userAge = inSS.nextInt();

      // Output parsed values
      System.out.println("First name: " + firstName);
      System.out.println("Last name: " + lastName);
      System.out.println("Age: " + userAge);

      return;
   }
}
```

```
First name: Amy
Last name: Smith
Age: 19
```

The program uses `import java.util.Scanner;` for access to the Scanner class. The statement `Scanner inSS = new Scanner(userInfo);` creates a Scanner object in which the associated input stream is initialized with a copy of myString. Then, the program can extract data from the scanner inSS using the family of next() methods (e.g., next(), nextInt(), nextDouble(), etc.).

A common use of string streams is to process user input line-by-line. The following program reads in the line as a String, and then extracts individual data items from that String.

Figure 9.4.2: Using a string stream to process a line of input text.

```
import java.util.Scanner;

public class ProcessInputText {
   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in); // Input stream for standard input
      Scanner inSS = null;                   // Input string stream
      String lineString = "";                // Holds line of text
      String firstName = "";                 // First name
      String lastName = "";                  // Last name
      int userAge = 0;                       // Age
      boolean inputDone = false;             // Flag to indicate next iteration

      // Prompt user for input
```

```
        System.out.println("Enter \"firstname lastname age\" on each line");
        System.out.println("(\"Exit\" as firstname exits).\n");

        // Grab data as long as "Exit" is not entered
        while (!inputDone) {

           // Entire line into lineString
           lineString = scnr.nextLine();

           // Create new input string stream
           inSS = new Scanner(lineString);

           // Now process the line
           firstName = inSS.next();

           // Output parsed values
           if (firstName.equals("Exit")) {
              System.out.println("   Exiting.");

              inputDone = true;
           }
           else {
              lastName = inSS.next();
              userAge = inSS.nextInt();

              System.out.println("   First name: " + firstName);
              System.out.println("   Last name: " + lastName);
              System.out.println("   Age: " + userAge);
              System.out.println();
           }
        }

        return;
     }
  }
```

```
Enter "firstname lastname age" on each line
("Exit" as firstname exits).

Mary Jones 22
   First name: Mary
   Last name: Jones
   Age: 22

Mike Smith 24
   First name: Mike
   Last name: Smith
   Age: 24

Exit
   Exiting.
```

The program uses `scnr.nextLine()` to read an input line from the standard input and copy the line into a String. The statement `inSS = new Scanner(lineString);` uses the Scanner's constructor to initialize the stream's buffer to the String lineString. Afterwards, the program extracts input from that stream using the next() methods.

Similarly, a new ***output string stream*** can be created that is associated with a string rather than with the screen (standard output). An output string stream is created using both the StringWriter and PrintWriter classes, which are available by including: `import java.io.StringWriter;` and `import java.io.PrintWriter;`

The StringWriter class provides a character stream that allows a programmer to output characters. The PrintWriter class is a wrapper class that augments character streams, such as StringWriter, with print() and println() methods that allow a programmer to output various data types (e.g., int, double, String, etc.) to the underlying character stream in a manner similar to System.out.

To create a PrintWriter object, the program must first create a StringWriter, passing the StringWriter object to the constructor for the PrintWriter. Once the PrintWrite object is created, a program can insert characters into that stream using print() and println(). The program can then use the StringWriter's ***toString()*** method to copy that buffer to a String.

Notice that the PrintWriter object provides the print() and println() methods for writing to the stream, and the StringWriter object provides the toString() method for getting the resulting String. The following example illustrates the use of StringWriter and PrintWriter classes.

Figure 9.4.3: Creating a String using a streams.

```
import java.util.Scanner;
import java.io.PrintWriter;
import java.io.StringWriter;

public class StringOutputStream {
   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);

      // Basic character stream for fullname
      StringWriter fullnameCharStream = new StringWriter();
      // Augments character stream (fullname) with print()
      PrintWriter fullnameOSS = new PrintWriter(fullnameCharStream);
      // Basic character stream for age
      StringWriter ageCharStream = new StringWriter();
      // Augments character stream (age) with print()
      PrintWriter ageOSS = new PrintWriter(ageCharStream);

      String firstName = ""; // First name
      String lastName = "";  // Last name
      String fullName = "";  // Full name (first and last)
      String ageStr = "";    // Age (string)
      int userAge = 0;       // Age

      // Promput user for input
      System.out.print("Enter \"firstname lastname age\": \n   ");
      firstName = scnr.next();
      lastName = scnr.next();
      userAge = scnr.nextInt();

      // Writes formatted string to buffer, copies from underlying char buffer
      fullnameOSS.print(lastName + ", " + firstName);
      fullName = fullnameCharStream.toString();
```

```
         // Output parsed input
         System.out.println("\n   Full name: " + fullName);

         // Writes int age as characters to buffer
         ageOSS.print(userAge);

         // Appends (minor) to object if less than 21, then
         // copies buffer into string
         if (userAge < 21) {
            ageOSS.print(" (minor)");
         }

         ageStr = ageCharStream.toString();

         // Output string
         System.out.println("   Age: " + ageStr);

         return;
      }
   }
```

```
Enter "firstname lastname age":
   Mary Jones 22

   Full name: Jones, Mary
   Age: 22

...

Enter "firstname lastname age":
   Sally Smith 14

   Full name: Smith, Sally
   Age: 14 (minor)
```

Participation Activity

9.4.1: Input/output string streams.

#	Question	Your answer
1	Define and initialize a Scanner variable named inSS that creates an input string stream using the String variable myStrLine.	
2	Define and initialize a PrintWriter variable named outSS that creates an output string stream using the underlying stream given by `StringWriter simpleStream = new StringWriter();`.	
3	Write a statement that copies the contents of an output string stream to a String variable called myStr. Assume the StringWriter and PrintWriter variables are called simpleStream and outSS respectively.	

Challenge Activity

9.4.1: Reading from a string.

Write code that uses the input string stream inSS to read input data from string userInput, and updat and userYear. Sample output if userinput is "Jan 12 1992":

```
Month: Jan
Date: 12
Year: 1992
```

```
3  public class StringInputStream {
4     public static void main (String [] args) {
5        Scanner inSS = null;
6        String userInput = "Jan 12 1992";
7        inSS = new Scanner(userInput);
8
9        String userMonth = "";
10       int userDate = 0;
11       int userYear = 0;
12
13       /* Your solution goes here  */
14
15       System.out.println("Month: " + userMonth);
16       System.out.println("Date: " + userDate);
17       System.out.println("Year: " + userYear);
18
19       return;
20    }
21 }
```

Run

Challenge Activity

9.4.2: Output using string stream.

Write code that inserts userItems into the output string stream itemsOSS until the user enters "Exit". space. Sample output if user input is "red purple yellow Exit":

```
red purple yellow
```

```
10        StringWriter itemCharStream = new StringWriter();
11        PrintWriter itemsOSS = new PrintWriter(itemCharStream);
12
13        System.out.println("Enter items (type Exit to quit):");
14        userItem = scnr.next();
15
16        while (!userItem.equals("Exit")) {
17
18           /* Your solution goes here  */
19
20           userItem = scnr.next();
21        }
22
23        userItem = itemCharStream.toString();
24        System.out.println(userItem);
25
26        return;
27     }
28  }
```

Run

Section 9.5 - File input/output

Sometimes a program should get input from a file rather than from a user typing on a keyboard. To achieve this, a programmer can create a new input stream that comes from a file, rather than the predefined input stream System.in that comes from the standard input (keyboard). That new input stream can then be used just like the familiar Scanner and System.in combination, as the following program illustrates. Assume a text file exists named myfile.txt with the contents shown (created for example using Notepad on a Windows computer or using TextEdit on a Mac computer).

Figure 9.5.1: Input from a file.

```
import java.util.Scanner;
import java.io.FileInputStream;
import java.io.IOException;

public class FileReadNums {
   public static void main (String[] args) throws IOException {
      FileInputStream fileByteStream = null; // File input stream
      Scanner inFS = null;                   // Scanner object
      int fileNum1 = 0;                      // Data value from file
      int fileNum2 = 0;                      // Data value from file

      // Try to open file
      System.out.println("Opening file myfile.txt.");
      fileByteStream = new FileInputStream("myfile.txt");
      inFS = new Scanner(fileByteStream);

      // File is open and valid if we got this far (otherwise exception thrown)
      // myfile.txt should contain two integers, else problems
      System.out.println("Reading two integers.");
      fileNum1 = inFS.nextInt();
      fileNum2 = inFS.nextInt();

      // Output values read from file
      System.out.println("num1: " + fileNum1);
      System.out.println("num2: " + fileNum2);
      System.out.println("num1+num2: " + (fileNum1 + fileNum2));

      // Done with file, so try to close it
      System.out.println("Closing file myfile.txt.");
      fileByteStream.close(); // close() may throw IOException if fails

      return;
   }
}
```

myfile.txt with variable number of integers:

```
5
10
```

```
Opening file myfile.txt.
Reading two integers.
num1: 5
num2: 10
num1+num2: 15
Closing file myfile.txt.
```

Six lines are needed for the new file input stream, highlighted above.

- The import statements `import java.io.FileInputStream;` and `import java.io.IOException;` enable the use of the ***FileInputStream*** and IOException classes respectively. The FileInputStream class, which is derived from InputStream, allows a programmer to read bytes from a file, and the IOException class provides mechanisms for exception throwing and handling, which are discussed

in more detail elsewhere.

- The statement `fileByteStream = new FileInputStream("myfile.txt");` creates a file input stream and opens the file denoted by the String literal for reading. Note that FileInputStream's constructor allows a programmer to specify a file using a String variable as well (e.g., `fileByteStream = new FileInputStream(fileStr);`).
- The FileInputStream class only supports a basic byte stream, and thus the statement `inFS = new Scanner(fileByteStream);` creates a new Scanner object using the fileByteStream object.
- Because of the high likelihood that the file fails to open, usually because the file does not exist or is in use by another program, the main() method definition contains a throws clause specifying that an exception of type IOException may be thrown within the method causing the program to terminate.
- If the statement `fileByteStream = new FileInputStream("myfile.txt");` does not throw an exception, the successfully opened input stream and scanner object can then be used to read from the file using the scanner's next() methods, e.g., using `num1 = inFS.nextInt();` to read an integer into num1.
- When done using the stream, the program closes the file (and input stream) using `fileByteStream.close()`.

A common error is a mismatch between the variable data type and the file data, e.g., if the data type is int but the file data is "Hello".

Try 9.5.1: Good and bad file data.

File input, with good and bad data: Create myfile.txt with contents 5 and 10, and run the above program. Then, change "10" to "Hello" and run again, observing the program terminate due to a runtime exception.

The following provides another example wherein the program reads items into an array. For this program, myfile.txt's first entry must be the number of numbers to read, followed by those numbers, e.g., 5 10 20 40 80 1.

Figure 9.5.2: Program that reads data from myfile.txt into an array.

```
import java.util.Scanner;
```

```
import java.io.FileInputStream;
import java.io.IOException;

public class FileReadNumsIntoArray {
   public static void main(String[] args) throws IOException {
      FileInputStream fileByteStream = null; // File input stream
      Scanner inFS = null;                   // Scanner object
      int[] userNums;                        // User numbers; memory allocated late
      int numElem = 0;                       // User-specified number of numbers
      int i = 0;                             // Loop index

      // Try to open file
      fileByteStream = new FileInputStream("myfile.txt");
      inFS = new Scanner(fileByteStream);

      // File is open and valid if we got this far (otherwise exception thrown)
      // Can use inFS stream via the Scanner object
      numElem = inFS.nextInt();               // Get number of numbers (first item)
      userNums = new int[numElem];            // Allocate enough memory for nums

      // Get numElem numbers. If too few, may encounter problems
      i = 1;
      while (i <= numElem) {
         userNums[i - 1] = inFS.nextInt();
         i = i + 1;
      }

      // Done with file, so try to close it
      fileByteStream.close(); // close() may throw IOException if fails

      // Print numbers
      System.out.print("Numbers: ");

      i = 0;
      while (i < numElem) {
         System.out.print(userNums[i] + " ");

         ++i;
      }

      System.out.println("");

      return;
   }
}
```

myfile.txt file contents:

```
5
10
20
40
80
1
```

```
Numbers: 10 20 40 80 1
```

A program can read varying amounts of data in a file by using a loop that reads until valid data is

unavailable or the end of the file has been reached, as follows. The ***hasNextInt()*** method returns true if an integer is available for reading. If the next item in the file is not an integer or if the previous stream operation reached the end of the file, the method returns false. The Scanner class offers multiple hasNext() methods for various data types such as int, double, String, etc..

Figure 9.5.3: Reading a varying amount of data from a file.

```
import java.util.Scanner;
import java.io.FileInputStream;
import java.io.IOException;

public class FileReadVaryingAmount {
   public static void main(String[] args) throws IOException {
      FileInputStream fileByteStream = null; // File input stream
      Scanner inFS = null;                   // Scanner object
      int fileNum = 0;                       // Data value from file

      // Try to open file
      System.out.println("Opening file myfile.txt.");
      fileByteStream = new FileInputStream("myfile.txt");
      inFS = new Scanner(fileByteStream);

      // File is open and valid if we got this far (otherwise exception thrown)
      System.out.println("Reading and printing numbers.");
      fileNum = inFS.nextInt();

      while (inFS.hasNextInt()) {
         System.out.println("num: " + fileNum);

         fileNum = inFS.nextInt();
      }
      System.out.println("num: " + fileNum);

      // Done with file, so try to close it
      System.out.println("Closing file myfile.txt.");
      fileByteStream.close(); // close() may throw IOException if fails

      return;
   }
}
```

myfile.txt with variable number of integers:

```
111
222
333
444
555
```

```
Opening file myfile.txt.
Reading and printing numbers.
num: 5
num: 10
num: 20
num: 40
num: 80
Closing file myfile.txt.
```

Similarly, a program may write output to a file rather than to the standard output, as shown below. The

program creates an object of type ***FileOutputStream***, which is a kind of (i.e., is derived from) OutputStream. Because an OutputStream only supports a basic byte output stream, a PrintWriter object is created that enables a programmer to use the print() and println() methods in order to write various data types to the file (in manner similar to using print() and println() methods for System.out).

Figure 9.5.4: Sample code for writing to a file.

```
import java.io.PrintWriter;
import java.io.FileOutputStream;
import java.io.IOException;

public class FileWriteSample {
   public static void main(String[] args) throws IOException {
      FileOutputStream fileByteStream = null; // File output stream
      PrintWriter outFS = null;               // Output stream

      // Try to open file
      fileByteStream = new FileOutputStream("myoutfile.txt");
      outFS = new PrintWriter(fileByteStream);

      // File is open and valid if we got this far (otherwise exception thrown)
      // Can now write to file
      outFS.println("Hello");
      outFS.println("1 2 3");
      outFS.flush();

      // Done with file, so try to close it
      fileByteStream.close(); // close() may throw IOException if fails

      return;
   }
}
```

myoutfile.txt with variable number of integers:

```
Hello
1 2 3
```

Participation Activity

9.5.1: File input/output.

#	Question	Your answer

	What is the error in the following code?
	`FileInputStream fbStream;`
	`Scanner inFS;`
	`int[] num;`
		The file stream is not big enough.

		The file stream has not been properly opened.

1	`int numElem = 0;`
	`int i = 0;`
	`inFS = new Scanner(fbStream);`
	`numElem = inFS.nextInt();`
	`num = new int[numElem];`	

		The nextInt() method cannot be used here.
		The code is fine.
2	Which statement opens a file outfile.txt given `FileOutputStream obFS = null;`	`obFS.FileOutputStream("outfile.txt");`
		`obFS(outfile.txt);`
		Declare a FileInputStream not a FileOutputStream.
		`obFS = new FileOutputStream("outfile.tx`
		`obFS("outfile.txt");`

3	Given the following code, which correctly initializes outFS to enable a programmer to write to outfile.txt using PrintWriter's print() methods.
	`FileOutputStream obFS =`
	`new`
	`FileOutputStream("outfile.txt");`
	`PrintWriter outFS = null;`
		`outFS(obFS);`

		`outFS = new PrintWriter(obFS);`
		`obFS = new PrintWriter(obFS);`
		`outFS = PrintWriter(obFS);`
		`outFS = FileOutputStream(obFS);`

4	Given the following code, which correctly writes "apples" to file outfile.txt?
	`FileOutputStream obFS =`
	`new`
	`FileOutputStream("outfile.txt");`
	`PrintWriter outFS =`
	`new PrintWriter(obFS);`
		`outfile.print("apples");`

		`obFS.print("apples");`
		`outFS.print("apples");`
		`PrintWriter.print(apples);`

```
FileOutputStream.print("apples");
```

Exploring further:

- Oracle's Java FileOutputStream class specification
- Oracle's Java PrintWriter class specification