AP Computer Science　　　　　　　　　Name: ______________________________
Chapter 9 Notes (3)

Strings

<u>Formating Numbers into Strings</u>:

As we've previously discussed, the easiest way to convert a number into a string is to concatenate that number with a string. For example:

```
int n = -123;
String s = "" + n;  // s gets the value "-123"
s = "n = " + n;     // s gets the value "n = -123"
double x = -1.23;
s = "" + x;         // s gets the value "-1.23"
```

Java offers two other ways to convert an `int` into a string.

<u>The first way</u> is to use the static method `toString(int n)` of the `Integer` class:

```
int n = -123;
String s = Integer.toString(n);    // s gets the value "-123";
```

The `Integer` class belongs to the `java.lang` package, which is automatically imported into all programs. It is called a *wrapper class* because it "wraps" around a primitive data type `int`: you can take an `int` value and construct an `Integer` object from it. "Wrapping" allows you to convert a value of a primitive type into an object. For example, you might want to hold integer values in a list represented by the Java library class `ArrayList`, and `ArrayList` only works with objects. `java.lang` also has the `Double` wrapper class for `double`s and the `Character` wrapper class for `char`s. `Integer` has a method `intValue`, which returns the "wrapped" `int` value, and a method `toString` that returns a string representation of this `Integer` object. For example,

```
Integer obj1 = new Integer(3);
Integer obj2 = new Integer(5);
Integer sum = new Integer(obj1.intValue() + obj2.intValue());
System.out.println(sum);
```

displays 8. Similarly, the `Double` class has a method `doubleValue` and the `Character` class has a method `charValue`.

For now, it is important to know that the `Integer`, `Double`, and `Character` classes offer several "public service" <u>static</u> methods. An overloaded version of `toString`, which takes one parameter, is one of them.

<u>The secondway</u> is to use the static method `valueOf` of the `String` class. For example:

```
int n = -123;
String s = String.valueOf(n);      // s gets the value “-123”;
```

Similar methods work for `double` values (using the `Double` wrapper class). For example:

```
double x = 1.5;
String s1 = Double.toString(x);    // s1 gets the value “1.5”;
String s2 = String.valueOf(x);     // s2 gets the value “1.5”;
```

For `double`s, though, the number of digits in the resulting string may vary depending on the value, and a `double` may even be displayed in scientific notation.

◇ ◇ ◇

It is often necessary to convert a `double` into a string according to a specified format. This can be accomplished by using an object of the `DecimalFormat` library class and its `format` method. First you need to create a new `DecimalFormat` object that describes the format. For example, passing the `“000.0000”` parameter to the `DecimalFormat` constructor indicates that you want a format with at least three digits before the decimal point (possibly with leading zeroes) and four digits after the decimal point. You use that format object to convert numbers into strings. We won’t go too deeply into his here, but your programs can imitate the following examples:

```
import java.text.DecimalFormat;
…
     // Create a DecimalFormat object specifying at least one digit
     // before the decimal point and 2 digits after the decimal pt:
     DecimalFormat money1 = new DecimalFormat(“0.00”);

     // Create a DecimalFormat object specifying $ sign
     // before the leading digit and comma separators:
     DecimalFormat money2 = new DecimalFormat(“$#,##0”);

     // Convert totalSales into a string using these formats:
     double totalSales = 12345678.9;
     String s1 = money1.format(totalSales);
          // s1 gets the value “12345678.90”
     String s2 = money2.format(totalSales);
          // s2 gets the value “$12,345,679” due to founding

     // Create a DecimalFormat object specifying 2 digits
     //   (with a leading zero, if necessary):
     DecimalFormat twoDigits = new DecimalFormat(“00”);

     // Convert minutes into a string using twoDigits:
     int minutes = 7;
     String s3 = twoDigits.format(minutes);
          // s3 gets the value “07”
```

If, for example, `totalSales` is 123.5 and you need to print something like

```
Total sales: 123.50
```

you could write

```
System.out.print("Total sales: " + money1.format(totalSales));
```

If `hours = 3` and `minutes = 7` and you want the time to look like `3:07`, you could write

```
System.out.print(hours + ":" + twoDigits.format(minutes));
```

Starting with the Java 5.0 release, `PrintStream` and `PrintWriter` objects (including `System.out` and text files open for writing) have a convenient method `printf` for writing formatted output to the console screen and to files. printf is an unusual method: it can take a variable number of parameters. The first parameter is always a format string, usually a literal string. The format string may contain fixed text and one or more embedded *format specifiers.* The rest of the parameters correspond to the format specifiers in the string. For example:

```
int month = 5, day = 19, year = 2007;
double amount = 123.5;
System.out.printf("Date: %02d/%02d/d  Amount: %7.2f\n",
                                        month, day, year, amount);
```

displays

```
Date: 05/19/2007    Amount: 123.50
```

Here `%02d` indicates that the corresponding output parameter (`month`, then `day`) must be formatted with two digits including a leading zero if necessary, `%d` indicates that the next parameter (`year`) should be an integer in default representation (with whatever sign and number of digits it might have); `%7.2f` indicates that the next parameter (`amount`) should appear as a floating-point number, right-justified in a field of width 7, with two digits after the decimal point, round if necessary. `\n` at the end tells `printf` to advance to the next line. The details of `printf` formatting are rather involved – refer to the Java API documentation.

The Java 5.0 release has also added an equivalent of `printf` for "writing" into a string. The static method `format` of the `String` class arranges several inputs into a formatted string and returns that string. For example:

```
int month = 5, day = 19, year = 2007;
double amount = 123.5;
String msg = String.format("Date: %02d/%02d/%d   Amount:  %7.2f",
                                   month, day, year, amount);
```

The above statement set `msg` to "`Date: 05/19/2007   Amount:   123.50`".

Here `%02d` indicates that the corresponding output parameter (`month`, then `day`) must be formatted with two digits including a leading zero if necessary; %d indicates that the next parameter (`year`) should be an integer in default representation (with whatever sign and number of digits it might have); `%7.2f` indicates that the next parameter (`amount`) should appear as a floating-point number, right-justified in a field of width 7, with two digits after the decimal point, rounded if necessary. `\n` at the end tells `printf` to advance to the next line. The details of `printf` formatting are rather involved – refer to the Java API documentation.

The Java 5.0 release has also added an equivalent of `printf` for "writing" into a string. The static method `format` of the `String` class arranges several input into a formatted string and returns that string. For example:

```
int month = 5, day = 19, year = 2007;
double amount = 123.5;
String msg = String.format("Date: %02d/%02d/%d    Amount: %7.2f",
                                                month, day, year, amount);
```

The above statements set msg to `"Date: 05/19/2007   Amount: 123.50"`.

Extracting Numbers from Strings:

The reverse operation – converting a string of digits (with a sign, if present) into an `int` value – can be accomplished by calling the static `parseInt` method of the `Integer` class. For example:

```
String s = "-123";
int n = Integer.parseInt(s);  // n gets the value -123
```

What happens if the string parameter passed to `parseInt` does not represent a valid integer? This question takes us briefly into the subject of Java *exception handling*.

If `parseInt` receives a bad parameter, it throws a `NumberFormatException`. You have already seen several occasions when a program "throws" a certain "exception" if it encounters some bug or unexpected situation. This exception, however, is different in nature from the other exceptions that we have experienced up to now, such as `NullPointerException`, `IllegalArgumentException`, or `StringIndexOutOfBoundsException`. Those exceptions are the programmer's fault: they are caused by mistakes in the program. When one of them is thrown, there is nothing to do but to terminate the program and report where the error occurred.

But a `NumberFormatException` may be caused simply by incorrect input from the user. The user will be very surprised if the program quits just because he types an `'o'` instead of a `'0'`. The program should handle such situations gracefully, and Java provides a special tool for that: the `try-catch-finally` statement. You can call `parseInt` "tentatively," within a `try` block, and "catch" this parameter type of exception within the `catch` block that follows. The `catch` block is executed only when an exception is thrown. It may be followed by the `finally` block that is always executed and therefore can perform the necessary clean-up. `try`, `catch`, and `finally` are Java reserved words. The code below shows how this may be used.

```
    Scanner input = new Scanner(System.in);
    int n = 0;

    while(n <= 0)
    {
        System.out.print("Enter a positive integer: ");
        String str = input.next();          // reads a token
        input.nextLine();              // skip the rest of the line
        try                       // try to extract an int from a str
        {
            n = Integer.parseInt(str);
        }
        catch (NumberFormatException ex)   // skip this if successful
        {
            System.out.println("*** Invalid input ***");
        }
        finally // either way execute this
        {
            if (n <= 0)
                System.out.println("Your input must be a positive
integer");
        }
    }

    // Process n:

    …
```

A similar method, `parseDouble` of the `Double` class, can be used to extract a `double` value from a string. For example:

```
    String s = "1.5";
    double x = Double.parseDouble(s);  // x gets the value 1.5
```