# Strings in Java

Strings are variables that hold words. For example:

*String sName = "Manjeet";*

Just like many other classes in Java, the String class has many methods to mess with.

**Checking equality?**

To check if two strings are equal, you have to use the "equals" method. For example

*if (sName.equals("Jimmy"){.....*

if this is true then you the program will execute what is within this if structure.

**String comparisons:**

If you need to check to see if one string precedes another, you can't use the "<" or ">" sign. You have to use the *compareTo()* String method. This method compares two strings lexicographically. The result is a negative integer if this String object lexicographically precedes the argument string. The result is a positive integer if this String object lexicographically follows the argument string. The result is zero if the strings are equal.

See sample code (compare_to_example.java)

**Primitive variables vs. "wrappers".**

Primitive variables are the types that we have been using: int, double, char. These are "old-school" variables that work much like variables have for decades.

Java comes with equivalent "wrappers" for each variable type: Integer, Double, Character. These "wrappers" are full "classes" that have many "power-tools" associated with them. You will have a better understanding of them when we work with Object code in a later skill set, but you can still use them without fulling understanding "how" they work.

**Let's check to see if the first letter of a word is capitalized: beween 'A' and 'Z' inclusive.**

(Strings.java)

```
String sName = "jimmy", sNew;
char ch; character CH;
ch = sName.charAt(0); // get the first letter of the name, and put it in ch
if (ch>='A' && ch<='Z')
 {
        System.out.println(sName +" is a valid name");
  }
 else
{
        System.out.println(sName +" is not a valid name");
         CH = Character.toUpperCase(ch);
        sNew = CH.toString();
        sNew = sNew.concat(sName.substring(1));
        System.out.println(sNew );
}
```

Why do we have to make use of a "new" String? Strings in Java are immutable. You can't pluck a character out of a String and replace it. You will learn more about StringBuilders in a later skill set. Here we chose to build a new String from the old one.

# String Console Input

## String input using a Scanner:

The following code shows how to get user input directly and place the input into a String variable directly. The scanner requires that you import java.util. (string_console_input.java)

```
import java.util.*; // this import is placed above your public class statement.
```

*// (the very top of your code)*

```
String sName;
```

```
Scanner sin = new Scanner(System.in);
System.out.println("Enter your name");
sName = sin.next();
System.out.println("Howdy " + sName);
```

## String input using a buffered reader:

The strength of the scanner is that you can define the type of variable that you are reading in. For example: *nextInt()* assumes integer input, *next()* will allow String input, and *nextDouble()* for doubles. The buffered reader facilitates String input. If you expect numeric input, you will have to convert the String to a numeric variable, whether that be an integer or double.

Console input is a bit more involved in Java than in C++. When Java was created, most user interaction was assumed to come from a graphical user interface. Here's how to get input from the user at the console level:

First, you must put the following line after your initial "package" line:

```
import java.io.*;
```

Just like C++, Java is a slim language that forces you to "import" packages of code, so that you only load the tools that are required for the job. As you can see, we are "importing" a bunch of functions found in the "java.io." package.

Your next step is to modify your "main" line. Since you will be accepting user input, and that can cause exception errors, you must add "throws IOException" at the end as follows:

(buffered_reader_example.java)

```
public static void main(String[] args) throws IOException
{
```

Next, you will have to create a "buffered reader" in order to accept user input:

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

In this line, you have created an instance of a BufferedReader (that is part of the java.io package) and you have named it "br" (short for buffered reader). The rest of the line is all dealing with the setup of the buffered reader. The next lines of code will allow you to get user input and display it:

```
String sName;

System.out.print ("Please enter your name: ");
sName = br.readLine ();
System.out.println("Hello " + sName );
```

First, declare a "String" type variable - variables that hold words.

The next line asks the user for their name.

The third line then uses the BufferedReader called "br" in order to "readline" the user's input into the sName variable.

The final line displays the user's name in order to show we were successful at getting the user's input.

When you run this program, you will see "Please enter your name" in the output window at the bottom of the screen. You will have to drag your cursor over to this window in order to input the information.

Which is better? The scanner is better if you jump between String input and ints. The Scanner is better if you want to separate String input - first name, followed by last name - each in its own variable. The buffered reader is great for reading in entire lines of input. Use the Scanner whenever possible, and the buffered reader only when it is necessary.

## Numeric Input

### Numeric input using a Scanner:

The following code shows how to get user input directly and place the input into an integer variable directly.  (numeric_input_scanner.java)

```
import java.util.*;
int nRad;
final double dPI = 3.14159;
double dArea;
Scanner sin = new Scanner(System.in);
System.out.println("Enter the radius");
nRad = sin.nextInt();
dArea = dPI *(Math.pow(nRad, 2));
System.out.println("The area is " + dArea);
```

### Numeric input using a Buffered Reader:

You can get the user to input numbers, but you will be challenged to do any math with their input since the buffered reader only loads the input as strings - words.

Therefore, you will have to convert the user's numeric input into an integer by using the "parseInt()" function. The following code shows you how to load in numbers from the console, do a little math, and then display it:

(numeric_input_buffered_reader.java)

```
public static void main(String[] args)    throws IOException
{
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    String sNum;
    int nNum, nSquare;
    System.out.print ("Please enter a number: ");
    sNum = br.readLine ();
    nNum = Integer.parseInt( sNum ); // this is the important part!!
    nSquare = nNum * nNum;
    System.out.println(nNum +" squared is " + nSquare );
}
```

Notice the line that "sNum" is converted into an integer.

Which is better? The BufferedReader takes the input as a String, and then you have to later convert it using the parseInt() method. This is a good method if your data input is not reliable. What happens if the user enters "two" rather than "2". The BufferedReader would allow you to facilitate this issue. The Scanner would crash if the user tries to input words when it is expecting an int.

## Control Flow Structures: IF

Control flow structures deal with directing the traffic. At the heart of it are looping and "if" structures. Looping entails repeating a set of instructions for a defined number of times, while "if" structures allow the program to branch off in different areas depending on the value of given variables.

The basic structure of "if":

```
if(condition)
{      // stuff
}
else if( next condition)
{      // other stuff
}
else
{      // stuff to do if the first two conditions aren't met
}
```

Notice that the condition that you are testing is inside round brackets. IF that condition is true, then the program will execute what is within the brace brackets that follow. Also notice that all actions within brackets are **indented**. This must be done so that your code is clear to read. It will work if you don't indent, but it is important to be read by others. For example:

(IfScanner.java)

```
import java.util.*;
public class ifscanner
{
      public static void main(String[] args)
      {
            Scanner sin = new Scanner(System.in);
            int nMark;
            System.out.print("Please enter your mark: ");
            nMark = sin.nextInt();
            if (nMark > 80)
            {
                  System.out.println("You got an A");
            }
            if (nMark > 49)
            {
                  System.out.println("You passed!!");
             }
            else
            {
                  System.out.println("You failed!!");
            }
      }
}
```

What is nice about “else if” structures is that once a test is met, then all further tests are bypassed. For example, if a student got a mark of 85, “You got an A” will be displayed. “You passed” will not, even though 85 is a mark over 49. Test this code out. What happens when you switch the order, so that you test for nMark>49 then you test for nMark>80?

A quick note on equality
In Java, you must be explicit about equality. In Java, the equal sign is referred to as the “assignment operator”; you assign a value to a variable. To check for equality, you need to use double equal signs. For example:

```
if(nMark==50)
{
      System.out.println("You barely passed!!" );
}
```

Part of predicting your output would be testing the above if statement with a single equals sign, therefore:

```
if(nMark=50)
{
      System.out.println("You barely passed!!" );
}
```

Test it out. Did it work? Good, now try checking for *nMark = 21.*
What happened?

Brace brackets are only required if more than one line is to be executed within the "if" structure.

For example:

```
if(nMark>80)
      System.out.println("You got an A!!" );
else if( nMark>49)
      System.out.println("You passed!!" );
else     System.out.println("You failed!!" );
```

If your program had to do 2 or more lines of code based on a student getting more than 80, then brace bracketes are required.

**Relational Operators:** relational operators will always return either true or false:
= = The double equal sign returns a value of true if the values on both sides of the equal signs are equal
!= not equal to
< Less Than and all the others

>= Greater than OR equal to
VB's "select case" structure is closely matched:

Logical Operators:
&& Logical AND (Conjunction)
|| Logical OR (Disjunction)
! Logical NOT: !bCondition

**Relational Precedence:**
Assess the following statement to find out whether it returns true of false given x=3, and y and z are 10:
if(x>5 && y>5 || z>5)

The first condition is false (x>5) and therefore the assessment of x>5 and y>5 is also false. But, since z>5 evaluates to true, the entire statement evaluates to true.

By wrapping the later conditions is brackets so that they are evaluated first, we will get a different result:
if(x>5 && (y>5 || z>5))
Adding brackets always allows the programmer to ensure they are in control of the order of assessing relations. For clarity's sake, the conditions in brackets should have been put at the beginning of the if statement - after all - they are being assessed first.

With the **switch** statement the following takes place:
1. The value of the selector is determined
2. The first instance of the value is found among the labels.
3. The statements following this value are executed.
4. break statements are optional. If a break statement occurs within these statements, then control is transferred to the first program statement following the entire switch statement.

```
switch(variable)
{
case C1:
{ // stuff
}
      break;
case C2:
// more stuff
      break;
default: // when all else fails…
      break;
}

switch( c )
{
case 'A':
        capa++;
case 'a':
        lettera++;
default :
     total++;
}
switch( i )
{case -1:   n++;
     break;
case 0 :
      z++;
break;
case 1 :
     p++;
      break;
}
```

A switch structure can only be used when assessing the equality of ints or chars. chars have to be placed in single quotations: case 'a'.

It is worth playing around with this structure, and see what happens when you omit the break statements. Why?

**ternary "if" structure also known as the "selection operator":**

Look at the following code:

*nMax = (nA > nB) ? nA : nB;*

If nA is greater than nB, then nMax becomes nA, otherwise, it becomes nB.

**Comparing floating point numbers:**

Due to the "floating fudge factor" discussed earlier, checking equality of floats or doubles takes a bit more thinking.

You have to check if the two numbers are "close enough". Here's how.

1. Define your "epsilon" value. Wikipedia defines epsilon as: an arbitrarily small positive quantity.

*final double EPSILON = 1E-14; // a very small number*

Populate the two double variables that you want to compare:

*double dA, dB;*

*if(Math.abs(dA-dB) <=EPSILON) System.out.println("Close enough");*