**The Main Differences:**

1. **Scanner**

* A simple text scanner which can parse primitive types and strings using regular expressions.
* A Scanner breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various next methods.

Example

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

prints the following output:

1 2 red blue

The same output can be generated with this code, which uses a regular expression to parse all four tokens at once:

String input = "1 fish 2 fish red fish blue fish"; Scanner s = new Scanner(input); s.findInLine("(\\d+) fish (\\d+) fish (\\w+) fish (\\w+)"); MatchResult result = s.match(); for (int i=1; i<=result.groupCount(); i++) System.out.println(result.group(i)); s.close(); `

1. **BufferedReader:**
   * Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.
   * The buffer size may be specified, or the default size may be used. The default is large enough for most purposes.

In general, each read request made of a Reader causes a corresponding read request to be made of the underlying character or byte stream. It is therefore advisable to wrap a BufferedReader around any Reader whose read() operations may be costly, such as FileReaders and InputStreamReaders. For example,

BufferedReader in = new BufferedReader(new FileReader("foo.in"));

will buffer the input from the specified file. Without buffering, each invocation of read() or readLine() could cause bytes to be read from the file, converted into characters, and then returned, which can be very inefficient. Programs that use DataInputStreams for textual input can be localized by replacing each DataInputStream with an appropriate BufferedReader.

Following are the differences between BufferedReader and Scanner

1. BufferedReader only read data but scanner also parse data.
2. you can only read String using BufferedReader, but you can read int, long or float using Scanner.
3. BufferedReader is older from Scanner,it exists from jdk 1.1 while Scanner was added on JDK 5 release.
4. The Buffer size of BufferedReader is large(8KB) as compared to 1KB of Scanner.
5. BufferedReader is more suitable for reading file with long String while Scanner is more suitable for reading small user input from command prompt.
6. BufferedReader is synchronized but Scanner is not, which means you cannot shre Scanner among multiple threads.
7. BufferedReader is faster than Scanner because it doesn't spent time on parsing on basis of the points we can select our choice.

StringBuffer is a thread-safe class

public final class StringBuffer extends AbstractStringBuilder implements Serializable, CharSequence { // .. skip .. public synchronized StringBuffer append(StringBuffer stringbuffer) { super.append(stringbuffer); return this; } // .. skip .. }

But StringBuilder is not thread-safe, thus it is faster to use StringBuilder if possible

public final class StringBuilder extends AbstractStringBuilder implements Serializable, CharSequence { // .. skip .. public StringBuilder append(String s) { super.append(s); return this; } // .. skip .. }

In some cases this is obsolete due to optimisations performed by the compiler, but the general issue is that code like:

string myString=""; for(int i=0;i<x;i++) { myString += "x"; }

will act as below (each step being the next loop iteration):

1. construct a string object of length 1, and value "x"
2. Create a new string object of size 2, copy the old string "x" into it, add "x" in position 2.
3. Create a new string object of size 3, copy the old string "xx" into it, add "x" in position 3.
4. ... and so on

As you can see, each iteration is having to copy one more character, resulting in us performing 1+2+3+4+5+...+N operations each loop. This is an O(n^2) operation. If however we knew in advance that we only needed N characters, we could do it in a single allocation, with copy of just N characters from the strings we were using - a mere O(n) operation.

StringBuffer/StringBuilder avoid this because they are mutable, and so do not need to keep copying the same data over and over (so long as there is space to copy into in their internal buffer). They avoid performing an allocation and copy proportional to the number of appends done by over-allocing their buffer by a proportion of its current size, giving amortized O(1) appending.

However its worth noting that often the compiler will be able to optimise code into StringBuilder style (or better - since it can perform constant folding etc.) automatically.

For simple concatenations like:

String s = "a" + "b" + "c";

It is rather pointless to use StringBuffer - as jodonnell pointed out it will be smartly translated into:

String s = new StringBuffer().append("a").append("b").append("c").toString();

**BUT** it is very unperformant to concatenate strings in a loop, like:

String s = ""; for (int i = 0; i < 10; i++) { s = s + Integer.toString(i); }

Using string in this loop will generate 10 intermediate string objects in memory: "0", "01", "012" and so on. While writing the same using StringBuffer you simply update some internal buffer of StringBuffer and you do not create those intermediate string objects that you do not need:

StringBuffer sb = new StringBuffer(); for (int i = 0; i < 10; i++) { sb.append(i); }

Actually for the example above you should use StringBuilder (introduced in Java 1.5) instead of StringBuffer - StringBuffer is little heavier as all its methods are synchronized

**Below is the main difference between these three most commonly used classes.**

* String class objects are immutable whereas StringBuffer and StringBuilder objects are mutable.
* StringBuffer is synchronized while StringBuilder is not synchronized.
* Concatenation operator "+" is internal implemented using either StringBuffer or StringBuilder.

Criteria to choose among **String**, **StringBuffer** and **StringBuilder**

* If the Object value is not going to change use String Class because a String object is immutable.
* If the Object value can change and will only be accessed from a single thread, use a StringBuilder because StringBuilder is unsynchronized.
* In case the Object value can change, and will be modified by multiple threads, use a StringBuffer because StringBuffer is synchronized.