David Greenstein
Monta Vista High School

String Class

- An object in the **String** class represents a string of characters (char's).

- The **String** class is in the **java.lang** package which is loaded automatically by the compiler.

- The **String** class has constructors just like most other classes.

```
String str = new String(“Hello”);
```

String Class (cont)

The **String** class is unlike other classes because:

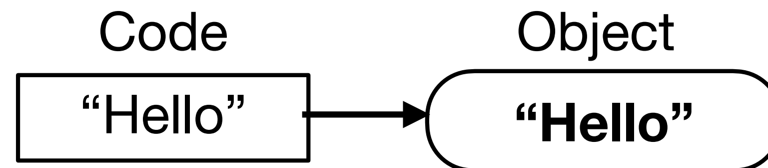
1. the **String** class has two operators, **+** and **+=**

2. the **String** class has <u>literal</u> objects denoted by **“”**

```
String str = new String(“Hello”);
str = str + new String(“ and “);
str += “goodbye”;
```

Literal String

- **Literal strings** are anonymous constant objects of the **String** class that are defined as text in double quotes.



- You can use a literal string to call **String** methods.

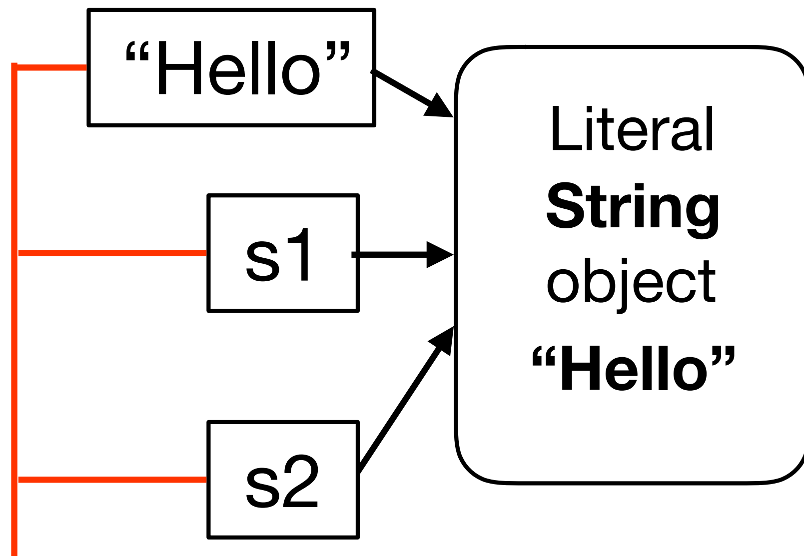
```
char a = "Hello".charAt(1);              // 'e'
String s = "Goodbye".substring(1,3); // "oo"
```

- **Literal strings** don't have to be constructed by your program; they are constructed and available before your code starts executing.

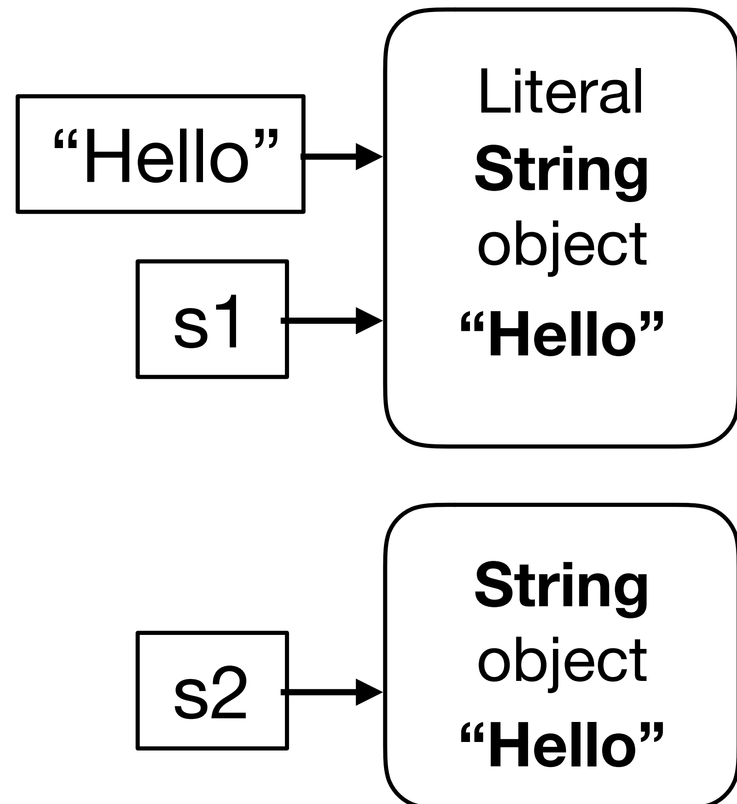
Literal Strings (cont)

- All similar literal strings point to the same anonymous **String** object.
- **new String** creates a brand new object.

```
String s1 = "Hello";
String s2 = "Hello";
```




```
String s1 = "Hello";
String s2 = new String("Hello");
```



Comparing Literal Strings

- Identifiers assigned to the same literal **String**s get the same pointer, so **==** produces true.

```
String s1 = "Hello", s2 = "Hello";
boolean p = (s1 == "Hello"); // true
boolean q = (s1 == s2);      // true
```

- **new String** creates a new object, so **==** with a literal produces false even if the string of characters match.

```
String s1 = "Hello";
String s2 = new String("Hello");
boolean p = (s1 == "Hello"); // true
boolean q = (s1 == s2);      // false
boolean r = (s2 == "Hello"); // false
```

Remember: == is comparing object references, not the string itself.

Escape Characters

The string text may include “escape” characters. For example:

- **** stands for ****
- **\n** stands for newline
- **\t** stands for tab
- **\”** stands for **”**

```
String s1 = “\tBiology”;
String s2 =  "C:\\jdk1.4\\docs";
String s3 = “\”Hello\” \n”;
```

Immutability

- Immutable objects cannot be changed.

- Immutable objects are convenient because several references can point to the same object safely.

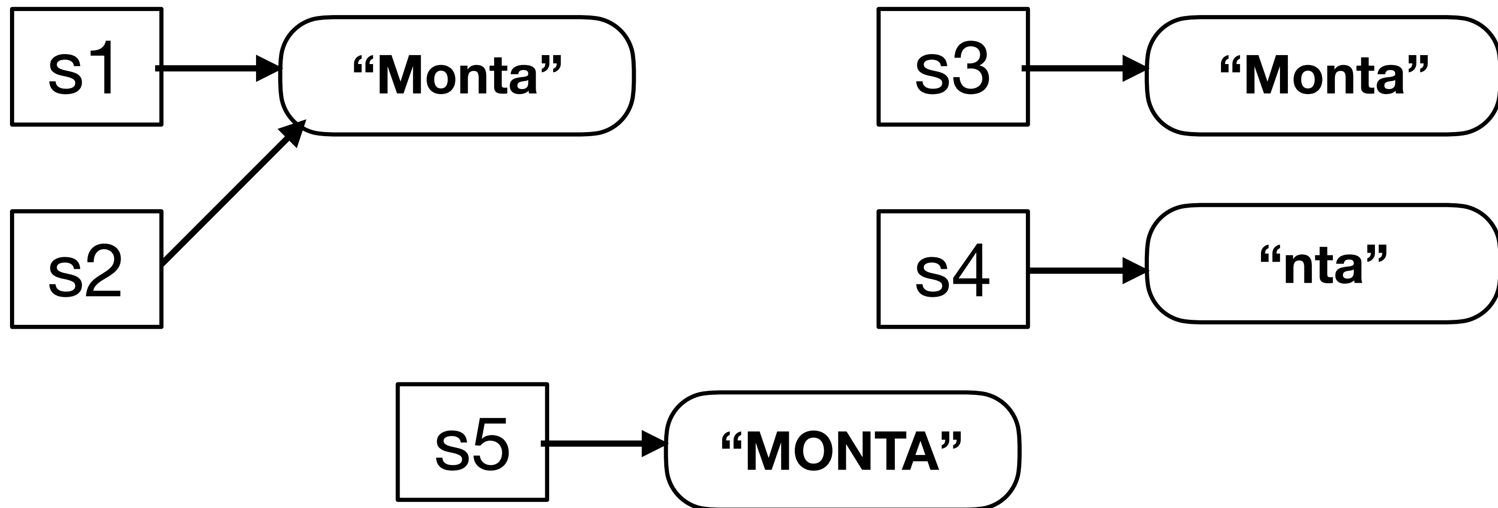
```
String s1 = “Hello”
String s2 = s1;
s1 = s1.toUpperCase(); // s2 does not change!
```

- The **java.lang** package has a number of immutable classes: **String**, **Integer**, **Double**, **Boolean**, etc.

Immutability (cont)

- **String** methods that modify the string create new objects

```
String s1 = "Monta";
String s2 = s1;                 // same reference
String s3 = new String(s1); // new object
String s4 = s1.substring(2); // new object
String s5 = s1.toUpperCase(); // new object
```

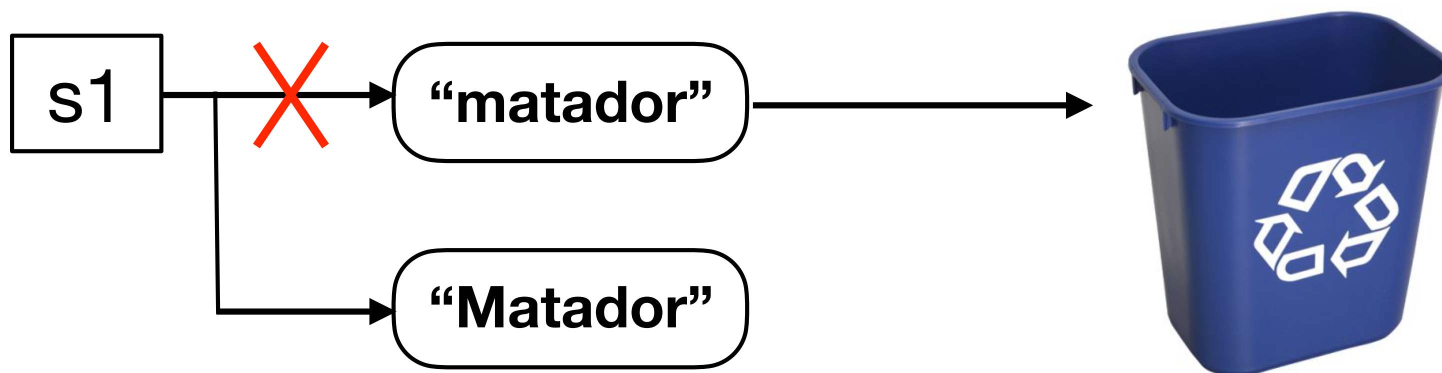



```
s1.substring(2); // Does not change s1!!!
```

Immutability (cont)

- **Advantage**: no need to copy — more efficient.
- **Disadvantage**: you need to create a new string and throw away the old one for every small change — less efficient.

```
String s1 = new String("matador");
s1 = Character.toUpperCases(s1.charAt(0)) +
                s1.substring(1);
```



Empty String

- An empty string has no characters; its length is 0.

- Two different ways to construct an **empty string object**.

```
String s1 = new String("");
String s2 = new String();
```

- Do not confuse an empty string object with an uninitialized string.

```
String msg;              // local variable
int x = msg.length(); // NullObject syntax error!
```

Methods - length, charAt

- **int length()** - returns the number of characters in the string

- **char charAt(int k)** - returns the character at index k inside the string

```
"Principal".length() returns 9
"Cupertino".charAt(5) returns 't'
```

Method - substring

- **substring** returns a new **String** object

```
String s1 = "halloween".substring(5);
```

→ "ween"

Returns a new string object starting with the character at index 5.

```
String s1 = "costume".substring(2,5);
```

→ "stu"

Returns a new string object starting with the character at index 2 up to but not including the character at index 5.

Method - substring (cont)

- An index one greater than the last index of the string allows the **substring** to capture the end of the string.

```
"ghost".substring(3,5);
```

→ "st"

- The **substring** can also create the empty string

```
"ghost".substring(5);
"ghost".substring(1,1);
```

→ ""
→ ""

- **substring** cannot accept an index out of bounds

```
"ghost".substring(6) // Index Out of Bounds
```

Method - Concatenation

```
String answer = s1 + s2;
```

Concatenates the strings s1 and s2.

```
String answer = s1.concat(s2);
```

Same as s1 + s2

```
s1 += s2;
```

Same as s1 + s2 with the result assigned to s1

Method - indexOf

Index

0	6	9	14	23

```
String date = "October 17, 2017 10:11:20 AM";
```

	Returns	
date.indexOf('r')	6	
date.indexOf('7')	9	
date.indexOf("17")	8	
date.indexOf("17", 10)	14	Searches from index 10
date.indexOf("2020")	-1	Not found
date.lastIndexOf('7')	15	Searches backward from end

Methods - Comparisons

- Most **String** comparisons should be done with **equals()** and **compareTo()**

```
boolean b = s1.equals(s2);
```

Returns **true** if the string **s1** is equal to **s2** character-for-character

```
boolean b = s1.equalsIgnoreCase(s2);
```

Same result as **equals()** but is case-blind

```
int diff = s1.compareTo(s2);
```

Returns the lexicographical “difference” **s1 - s2**

```
int diff = s1.compareToIgnoreCase(s2);
```

Returns the lexicographical “difference” **s1 - s2** but case-blind

Methods - Replacements

- **String** replacement methods return a new **String** object

```
String s2 = s1.trim();
```

Returns a new **String** object with whitespace characters removed from both ends of the string. Embedded whitespace is untouched.

```
String s2 = s1.replace(oldChar, newChar);
```

Returns a new **String** object in which every occurrence of `oldChar` is replaced by `newChar`

```
String s2 = s1.toUpperCase();
String s3 = s1.toLowerCase();
```

Returns a new **String** object that has all uppercase or lowercase characters of the original string

```
s1.toUpperCase() // Does not change s1!!!
```

Numbers to Strings

- There are four ways to convert a number into a string.

1. Concatenate a number with an empty string.

```
String s = "" + number;
```

2. Use the wrapper class of the number to convert to string.
   Integer and **Double** are the "wrapper" classes from **java.lang**.

```
String s1 = Integer.toString(intNum);
String s2 = Double.toString(dblNum);
```

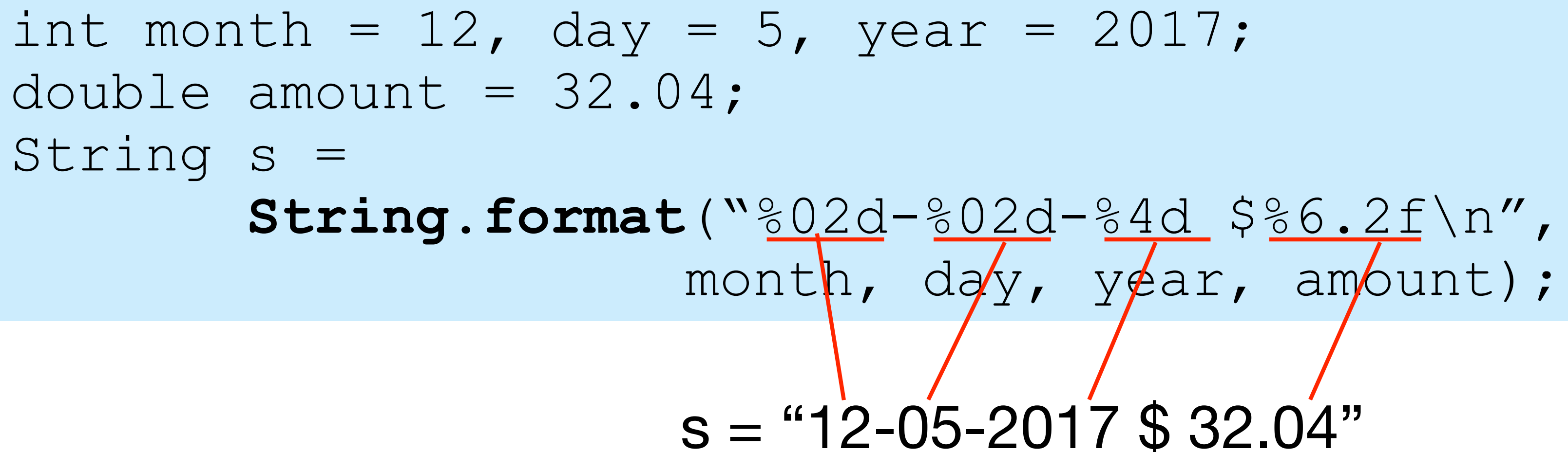
3. Use the **String** class **valueOf()** method.

```
String s = String.valueOf(num);
```

Numbers to Strings

- There are four ways to convert a number into a string.

4. Use the **String** class' **format()** method.



```
int month = 12, day = 5, year = 2017;
double amount = 32.04;
String s =
        String.format("%02d-%02d-%4d $%6.2f\n",
                      month, day, year, amount);
```

s = "12-05-2017 $ 32.04"

- Numbers are also converted in a similar way when using **printf()**.

```
System.out.printf("%02d-%02d-%4d $%6.2f\n",
                  month, day, year, amount);
```

Strings to Numbers

```
String s1 = "-123", s2 = "123.45";
int n = Integer.parseInt(s1);
double x = Double.parseDouble(s2);
```

- Numeric “wrapper” classes in **java.lang** have **parseXXX()** method to convert a **String** to a number.
  - Wrapper classes: **Byte**, **Short**, **Integer**, **Long**, **Double**, **Float**
- These methods throw a **NumberFormatException** if the string does not represent a valid number. Use **try-catch** when converting.

Character Methods

- **java.lang.Character** is a “wrapper” class that represents characters as objects.

- **Character** has several useful static methods that determine the type of a character.
  - **isLetter(char c)**
  - **isDigit(char c)**
  - **isLetterOrDigit(char c)**
  - **isUpperCase(char c)**
  - **isLowerCase(char c)**
  - **isWhitespace(char c)**

- **Character** also has methods that convert a letter to uppercase, lowercase, or to a **String**.
  - **toLowerCase(char c)**
  - **toUpperCase(char c)**
  - **toString(char c)** returns a String

StringBuffer and StringBuilder

- **StringBuffer** and **StringBuilder** classes are mutable versions of the **String** class.
  - Changes to the string affect the original object.
  - Advantage: More efficient in memory allocation.
  - Disadvantage: Changes affect all references to the string.

- **StringBuffer** and **String** are thread-safe classes.
  - Threads are asynchronous programs that run concurrently. (eg. Timers, GUIs)
  - If two or more threads try to change the same object, only one gets to change it. (thread-safe)

- **StringBuilder** is not thread-safe.
  - If two or more threads try to change the same object, then unexpected (meaning bad) things could happen to the object.

StringBuffer and **StringBuilder** are <u>not used</u> in this course.

Questions?