Java Fundamentals II

Michael C. Hackett

Assistant Professor, Computer Science

Lecture Topics

- Converting Primitives

- Arithmetic Operators
  - Precedence Rules
  - Augmented Assignment
  - Mixed Number Operations

- Math Object

- Strings
  - Concatenation and Appending
  - Parsing Integers and Doubles
  - Escape Sequences
  - Formatted Printing

Colors/Fonts

- Local Variable Names — **Brown**
- Primitive data types — **Fuchsia**
- Literals — **Blue**
- Keywords — **Orange**
- Object names — **Green**
- Operators/Punctuation — **Black**
- Field Names — **Lt Blue**
- Method Names — **Purple**
- Parameter Names — **Gold**
- Comments — **Gray**
- Package Names — **Pink**

Source Code — **Consolas**
Output — `Courier New`

Converting Primitives - Rank

- double   Highest rank
- float
- long
- int
- short
- byte   Lowest rank

- Ranking is important when you need to convert one data type to another.

- Higher ranked types can easily store lower ranked type since they are bigger and will have enough space to hold the value of the lower ranked type.

- Lower ranked types may not be big enough to hold the data that is stored in a higher ranked type.

Converting Primitives - Widening

- The process of copying the data of a lower ranked type into a higher ranked type is called **widening**.

- No real issues with this. You can just assign the lower ranked value into a variable of a higher rank.

```
double someDouble;
int someInt = 34653;
someDouble = someInt;
```

- The higher-ranked data type will have enough room to accommodate the value of the lower-ranked type.

- Remember: you can't change a variable's type after it has been declared.

Converting Primitives - Narrowing

- The process of moving the data of a higher ranked type into a lower ranked type is called *__narrowing__*.

```
double someDouble = 453.87;
int someInt;
someInt = (int)someDouble;
```

Put the desired data type, in parenthesis, before the variable name to *__typecast__* the value as that type.

Be careful! Narrowing can lead to loss of precision!
In the example above, the value stored at the memory location referenced by someInt is 453 NOT 453.87

Converting Primitives - Narrowing

- Consider the code below. An int value, 4500, is narrowed to a byte. However, a byte can only hold up to 127. What happens?

```
int someInt = 4500;
byte someByte;
someByte = (byte)someInt;
```

Converting Primitives - Narrowing

```
int someInt = 4500;
byte someByte;
someByte = (byte)someInt;
```

- 4500, in binary (and in memory,) looks like this: 1000110010100

- A byte can only hold eight bits: 10001 10010100

  These bits will be ignored.
  The technical term is ***overflow***

- The binary value of the data in memory referenced by someByte is now 10010100

- This binary value, when converted to a signed decimal integer is -62 (Way off from 4500)

Arithmetic Operators

- Addition: **+**
- Subtraction: **-**
- Multiplication: *****
- Division: **/**
- Modulo Division: **%**

Addition

```
int number1 = 6;
int number2 = 5;
int sum = number1 + number2;
```

- The memory location referenced by the variable sum is initialized with the value 11.

Subtraction

```
int number1 = 6;
int number2 = 5;
int difference = number1 - number2;
```

- The memory location referenced by the variable difference is initialized with the value -1.

Multiplication

```
int number1 = 6;
int number2 = 5;
int product = number1 * number2;
```

- The memory location referenced by the variable product is initialized with the value 30.

Division

```
int number1 = 8;
int number2 = 4;
int quotient = number1 / number2;
```

- The memory location referenced by the variable quotient is initialized with the value 2.

Division (Another example)

```
int number1 = 3;
int number2 = 4;
int quotient = number1 / number2;
```

- The memory location referenced by the variable quotient is initialized with the value 0.

- 3 / 4 equals 0.75

- ints cannot represent fractional numbers, so the fractional portion is dropped and we are left with zero.

Mod Division

- Finds the remainder of a division.

```
int number1 = 11;
int number2 = 4;
int remainder = number1 % number2;
```

- The memory location referenced by the variable remainder is initialized with the value 3.

- "11 divided by 4 is 2 <u>with a remainder of 3</u>"

Combined Assignment Operators

```
int myNumber = 11;

myNumber += 4;

myNumber -= 5;

myNumber *= 2;

myNumber /= 4;

myNumber %= 2;
```

Equivalent to:

```
myNumber = myNumber + 4;

myNumber = myNumber - 5;

myNumber = myNumber * 2;

myNumber = myNumber / 4;

myNumber = myNumber % 2;
```

Operator Precedence

- P[MD%][AS] (left to right)
- Multiplication, Division, Mod Division – same priority
- Addition, Subtraction – same priority

```
int num1 = 13;
int num2 = 5;
int num3 = 3;
int num4 = 2;

int answer = (num1 % num2 * num2) / num3 - num3 + num4;
```

- What is the value at the memory location referenced by the variable answer?

Mixed Number Operations

- Special rules apply when performing arithmetic operations on numbers of different types. For example, adding a byte and a double together or dividing an int by a short. *What data type is the result of that arithmetic*?

- Arithmetic operations performed only on a combination of *bytes, shorts, and ints* **ALWAYS return an int**!

- Arithmetic operations performed with *longs, floats, or doubles* **return with the highest ranked data type**.

Mixed Number Operations

```
byte byte1 = 10;
byte byte2 = 15;
byte byte3;
byte3 = byte1 + byte2;
```

- This will not work! It is easy to forget and think that two bytes added, divided, multiplied, etc will return an answer of type byte. It won't!

- Arithmetic operations performed only on a combination of *bytes, shorts, and ints* **ALWAYS return an int**!

```
byte byte1 = 10;
byte byte2 = 15;
int int1;
int1 = byte1 + byte2;
```
This will work.

Mixed Number Operations

- Why doesn't the addition of two bytes result in a byte?

```
byte byte1 = 100;
byte byte2 = 100;
int sumOfBytes;
sumOfBytes = byte1 + byte2;
```

- The sum of byte1 and byte2 equals 200.

- 200 is too large for a byte; 127 is the maximum.

- To avoid problems like this, any operations involving only bytes, shorts, and ints **always** returns an int.

Mixed Number Operations

```
byte byte1 = 10;
short short1 = 15;
short short2;
short2 = byte1 + short1;
```

- This will not work! It is easy to forget and think that since a short is larger than a byte that the answer will be a short. It won't!

- Arithmetic operations performed only on a combination of *bytes, shorts, and ints* **ALWAYS return an int**!

```
byte byte1 = 10;
short short1 = 15;
int int1;
int1 = byte1 + short1;
```
This will work.

Mixed Number Operations

```
byte byte1 = 10;
int int1 = 15;
int int2;
int2 = byte1 + int1;
```

- This will work!

- Arithmetic operations performed only on a combination of *bytes, shorts, and ints* **ALWAYS return an int**!

Mixed Number Operations

First Operand	Second Operand	Resulting Type
byte, short, int	byte, short, int	int
byte, short, int, long	long	long
byte, short, int, long, float	float	float
byte, short, int, long, float, double	double	double

Arithmetic when printing output

- You can preform arithmetic within a print/println method call.
  - The values referenced by the variables will not be changed.
  - The arithmetic is performed first; the result of the arithmetic is then passed to the print/println method.

```java
int totalAdults = 20;
int totalChildren = 22;
System.out.println(totalAdults + totalChildren);
System.out.println(totalAdults);
System.out.println(totalChildren);
```

```
42
20
22
```

Math object

- Provides access to mathematical functions not provided by default (like addition and subtraction).

- Doesn't need to be imported.

- Doesn't need to be instantiated.

Math object – Square Root

- Square Root function returns a **double**; Parameter can be any numeric type.

```java
int myNumber;
double squareRoot;
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter a number: ");
myNumber = Integer.parseInt(keyboard.nextLine());
squareRoot = Math.sqrt(myNumber);
System.out.println("The square root is: " + squareRoot);
```

```
Enter a number: 16
The square root is: 4.0
```

Math object – Exponents

- Pow(er) function returns a **double**

```
double base = 2;
int exponent = 3;
double result;
result = Math.pow(base, exponent);
System.out.println("The result is: " + result);
```

```
The result is: 8.0
```

Types of Rounding Functions

- A *round function* will round a fraction up or down to the nearest whole number.
  - Greater than .5 -> Rounded up
  - Less than .5 -> Rounded down
  - Exactly .5 -> Round down if even, round up if odd
    - 20.6 -> 21
    - 20.4 -> 20
    - 20.5 -> 20
    - 21.5 -> 22
- A *floor function* will round a fraction down.
  - Fractional is irrelevant.
  - 45.9 -> 45
- A *ceiling function* will round a fraction up.
  - Fractional is irrelevant.
  - 32.1 -> 33

Math object – Rounding

- Round method takes a **double or float.**

- Returns a **long** if double.

- Returns an **int** if float.

Math object – Rounding

```java
double originalNumber;
double roundedNumber;

Scanner keyboard = new Scanner(System.in);
System.out.print("Enter a decimal: ");
originalNumber = Double.parseDouble(keyboard.nextLine());
roundedNumber = Math.round(originalNumber);

System.out.println("The rounded number is: " + roundedNumber);

Enter a decimal: 25.6
The rounded number is: 26.0
```

Math object – Rounding Up

- Ceiling method takes a **double or float.**

- Returns a **double**.

Math object – Rounding Up

```java
double originalNumber;
double roundedNumber;

Scanner keyboard = new Scanner(System.in);
System.out.print("Enter a decimal: ");
originalNumber = Double.parseDouble(keyboard.nextLine());
roundedNumber = Math.ceil(originalNumber);

System.out.println("The rounded number is: " + roundedNumber);

Enter a decimal: 25.2
The rounded number is: 26.0
```

Math object – Rounding Down

- Floor method takes a **double or float.**

- Returns a **double**.

Math object – Rounding Down

```java
double originalNumber;
double roundedNumber;

Scanner keyboard = new Scanner(System.in);
System.out.print("Enter a decimal: ");
originalNumber = Double.parseDouble(keyboard.nextLine());
roundedNumber = Math.floor(originalNumber);

System.out.println("The rounded number is: " + roundedNumber);
```

```
Enter a decimal: 25.9
The rounded number is: 25.0
```

Math object methods

Method	Return Type	Description	Possible Exceptions
sqrt(double/float/long/int)	double	Returns the square root of the supplied number.	None
pow(double, double)	double	Returns the first parameter raised to the power of the second parameter.	None
round(double/float)	long if double; int if float	Rounds the supplied number to the nearest whole number and returns it.	None
ceil(double/float)	double	Rounds the supplied number up to the nearest whole number and returns it.	None
floor(double/float)	double	Rounds the supplied number down to the nearest whole number and returns it.	None
abs(double/float/long/int)	double	Returns the absolute value of the supplied numbers.	None
log(double)	double	Returns the natural logarithm of the supplied number.	None
toDegrees(double)	double	Returns the supplied number (radians) in degrees.	None
toRadians(double)	double	Returns the supplied number (degrees) in radians.	None

There are more than these. For a complete list go to: https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html

String Concatenation

- ***Concatenation*** is the process of joining Strings together into one String.

  - This is not the same as *appending*. When you concatenate Strings together, their individual values are not altered.

- Two ways to "concat" Strings:

  - Using the String object's concat method.
  - Using the addition operator.

Concat method

- A String's concat method returns the value of its String with the String value passed as a parameter added to it.

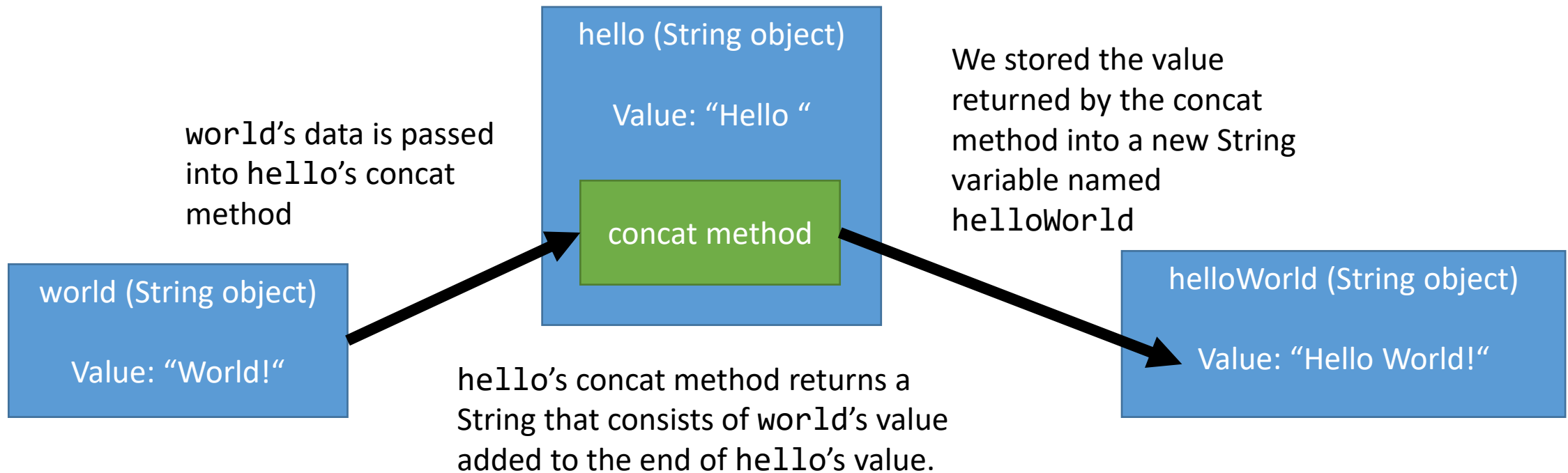- The values of the two Strings are not permanently altered.

```java
String hello = "Hello ";
String world = "World!";
String helloWorld = hello.concat(world);
System.out.println(helloWorld);
```

```
Hello World!
```

Note the values of the String variables hello and world **do not change**.

Concat method – What is actually happening?

```
String hello = "Hello ";
String world = "World!";
String helloWorld = hello.concat(world);
```

world's data is passed into hello's concat method

hello (String object)

Value: "Hello "

concat method

We stored the value returned by the concat method into a new String variable named helloWorld

world (String object)

Value: "World!"

helloWorld (String object)

Value: "Hello World!"

hello's concat method returns a String that consists of world's value added to the end of hello's value.

String Concatenation (Addition operator)

```
String hello = "Hello ";
String world = "World!";
String helloWorld = hello + world;
System.out.println(helloWorld);
```

```
Hello World!
```

Note the values of the String variables hello and world **do not change**.

String Concatenation

- Literals can be concatenated with the values of String variables.

```java
String hello = "Hello ";
String world = "World!";
System.out.println(hello + world
                    + " Have a great day!");
```

```
Hello World! Have a great day!
```

The variables hello, world, and the literal " Have a great day!" are concatenated together as one String. This one String is then passed to the System.out.println to be printed.

String Concatenation

- You can concatenate primitives into a String.

```java
int days = 31;
System.out.println("There are " + days
                        + " in January.");
```

```
There are 31 days in January
```

Appending to Strings

- ***Appending*** is the process of joining Strings together into one String that overwrites the original.
  - Unlike concatenation, appending alters values of a String.

- No method for appending.
  - You must concatenate and assign the new String value to the original.

Appending to Strings

- Concatenate to create a new String value.
- Assign the new value to the original variable.

```java
String greeting = "Have a good ";
greeting = greeting + "day!";
System.out.println(greeting);
```

```
Have a good day!
```

The above overwrites the original data, "Have a good ", with the value "Have a good day!"

Appending to Strings

- You can append primitives (variables or literals).

```
String farewell = "See you in ";
farewell = farewell + 2 + " days!";
System.out.println(farewell);
```

```
See you in 2 days!
```

Using Combined Assignment to Append

- You can use the addition combined operator (+=) to append to a String.

```java
String farewell = "See you in ";
farewell += "two" + " days!";
System.out.println(farewell);
```

```
See you in two days!
```

Converting a String to all UPPERCASE

- The toUpperCase method converts the String to a version of itself in all caps.

- Takes no parameters.

```
String hello = "Hello World!";
hello = hello.toUpperCase();
System.out.println(hello);
```

```
HELLO WORLD!
```

Note that we assign hello.toUpperCase() back to the String variable hello. In other words, we are replacing the original value "Hello World!" with "HELLO WORLD!" As you will see in the next slide, the toUpperCase() method doesn't actually change the String's value, it just returns the uppercase version of its data.
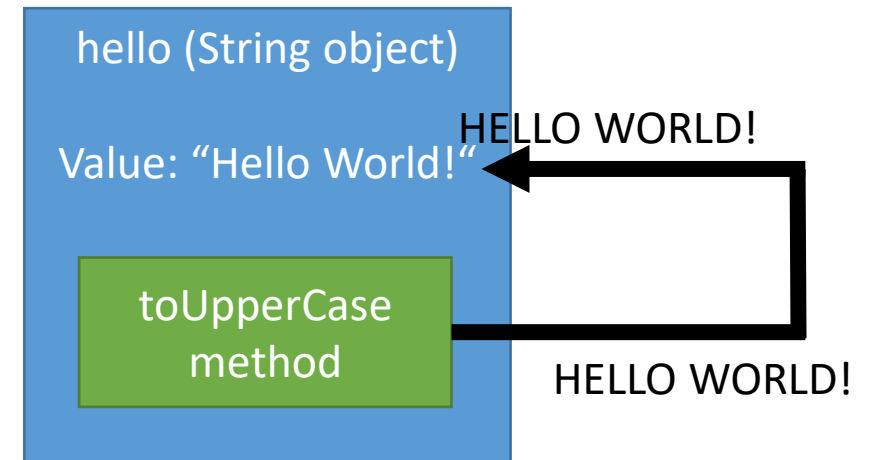
What is actually happening?

`String hello = "Hello World!";`

`hello.toUpperCase();`
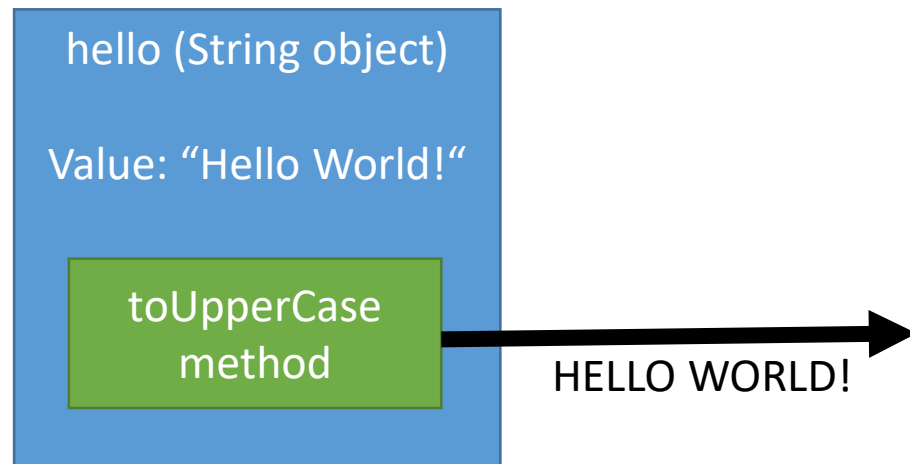
hello (String object)

Value: "Hello World!"

toUpperCase method

HELLO WORLD!

Doesn't actually change the String's value.

`hello = hello.toUpperCase();`

hello (String object)

HELLO WORLD!

Value: "Hello World!"

toUpperCase method

HELLO WORLD!

Here, we overwrite hello's initial value by assigning the result of the toUpperCase method back to the String itself.

Converting a String to all UPPERCASE

```java
String hello = "Hello World!";
System.out.println(hello.toUpperCase());
System.out.println(hello);
```

```
HELLO WORLD!
Hello World!
```

The toUpperCase() method doesn't actually change the String's value, it just returns the uppercase version of its data. When we print the value of hello again, it still has its original form ("Hello World!")

Converting a String to all lowercase

- The toLowerCase method converts the String to a version of itself in all lowercase.

- Takes no parameters.

```java
String hello = "Hello World!";
hello = hello.toLowerCase();
System.out.println(hello);
```
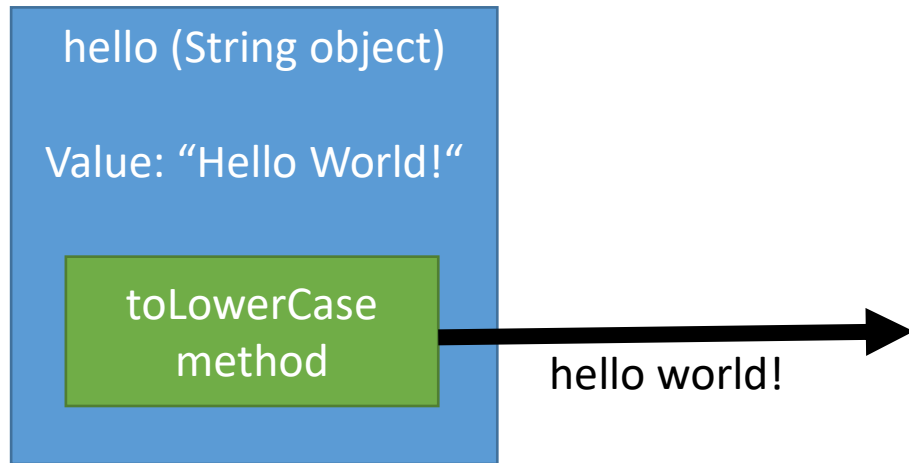
```
hello world!
```

Just like the toUpperCase() method, the toLowerCase() method doesn't actually change the String's value, it just returns the lowercase version of its data. In this example, we replaced the original value "Hello World!" with "hello world!"
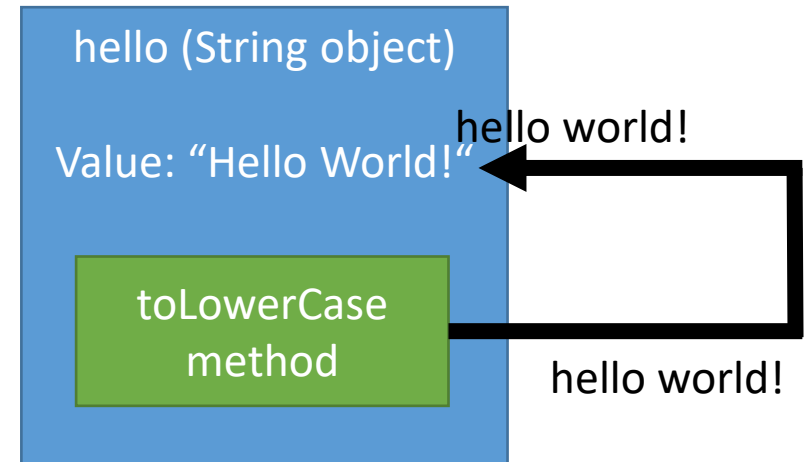
What is actually happening?

`String hello = "Hello World!";`

`hello.toLowerCase();`

hello (String object)

Value: "Hello World!"

toLowerCase method

hello world!

Doesn't actually change the String's value.

`hello = hello.toLowerCase();`

hello (String object)

hello world!

Value: "Hello World!"

toLowerCase method

hello world!

Here, we overwrite hello's initial value by assigning the result of the toLowerCase method back to the String itself.

String methods*

Method	Return Type	Description	Possible Exceptions
concat(String)	String	Concatenates the supplied String to the end of itself, returning this new value. Does not alter the original value.	None
toLowerCase()	String	Converts itself to all lowercase characters, returning this new value. Does not alter the original value.	None
toUpperCase()	String	Converts itself to all uppercase characters, returning this new value. Does not alter the original value.	None

*There are more than just these methods. We will use others as the semester progresses.

Integer object

- The Integer object is a special type of object known as a "wrapper" object (more on this later in the semester).
  - Essentially, the Integer object has an int inside of it... or, "wraps" around the int.
- A primitive like int doesn't have the capability of having methods. However, having it wrapped in an object will add that capability.

- The Integer object has a method called parseInt that takes in a String as a parameter, attempts to convert that String's value as a number, and returns it in int form.

Parsing Integers from Strings

```java
String ten = "10";
int myInt = ten + 15;
```

- The above code will not work! You cannot perform arithmetic with Strings, even if the String's characters are numbers.

- You also cannot directly assign a String to a numeric primitive like an int, float, etc even if the String's value is a number.

- Numbers must be *parsed* out of a String before you can use them as numeric values.

Parsing Integers from Strings

```java
String ten = "10";
int myInt = Integer.parseInt(ten);
myInt += 10;
System.out.println("Value of myInt plus 10 = "
                   + myInt);
```

```
Value of myInt plus 10 = 20
```

Double object

- Similar to how the Integer object wraps around an int, the Double object is a wrapper object for the double primitive data type.

- Also like the Integer object, the Double object has a method called parseDouble that takes in a String as a parameter, attempts to convert that String's value to a number, and returns it in double form.

Parsing Doubles from Strings

```java
String tenFive = "10.5";
double myDouble = Double.parseDouble(tenFive);
System.out.println("Value of myDouble plus 1 = "
                   + (myDouble + 1));
```

```
Value of myDouble plus 1 = 11.5
```

NumberFormatException

- A **NumberFormatException** is a runtime exception that will occur when you try to convert a String that isn't a number into a number.

```java
String letters = "abcd";
double myDouble = Double.parseDouble(letters);
System.out.println(myDouble);
```

```
Exception in thread "main" java.lang.NumberFormatException: For input string:
"abcd"
at sun.misc.FloatingDecimal.readJavaFormatString(Unknown Source)
at sun.misc.FloatingDecimal.parseDouble(Unknown Source)
at java.lang.Double.parseDouble(Unknown Source)
at test2.main(test2.java:8)
```

Integer and Double object methods*

Integer object

Method	Return Type	Description	Possible Exceptions
parseInt(String)	int	Returns the int value of the supplied String	NumberFormatException

Double object

Method	Return Type	Description	Possible Exceptions
parseDouble(String)	double	Returns the double value of the supplied String	NumberFormatException

*There are more than just these methods. We may use others as the semester progresses.

Escape Sequences

- \n – New Line
- \t - Tab
- \" – Double Quote
- \\ - Backslash

- There are more, but we will only be working with these few.

Escape Sequences - \n

- \n inserts a carriage return (or starts a new line)

```
System.out.print("Hello \nWorld");
```

```
Hello
World
```

Escape Sequences - \t

- Inserts one tabspace.

```
System.out.print("Hello \tWorld");
```

```
Hello     World
```

Escape Sequences - \"

- Inserts a double quote character
  - Without this, the compiler will interpret the " as the start/end of a String literal.

```
System.out.print("\"Hello\" World");
```

"Hello" World

Escape Sequences - \\

- Inserts a backslash character.
  - The single backslash indicates the start of an escape sequence to the compiler.
  - So, the backslash character itself needs to be escaped.

```
System.out.print("Hello \\ World");
```

```
Hello \ World
```

Formatted Printing - System.out.printf()

- The print and println methods print output with no formatting.

- The printf method allows greater control of how numbers are printed.

Formatted Printing - System.out.printf()

- Format specifiers begin with % and end with a converter character.
  - The converter character indicates the type of parameter to be formatted.

- Format specifiers are typed directly into a String literal.

```java
int age = 75;
System.out.printf("The value of age is %d", age);
```

```
The value of age is 75
```

Format Specifiers - %d

- Indicates the corresponding parameter is a decimal (base 10) integer (byte, short, int, long types.)

```java
int age = 75;
int age2 = 65;
System.out.printf("The ages are %d and %d", age, age2);
```

```
The ages are 75 and 65
```

Flags - %d

- Flags are optional an specify how the parameter is to be formatted.
- Inserted between the % and converter character.
- , flag - Inserts commas

```
int lotteryJackpot = 2500000;
System.out.printf("The jackpot is $%,d", lotteryJackpot);
```

```
The jackpot is $2,500,000
```

Format Specifiers - %f

- Indicates the corresponding parameter is a floating point decimal (float or double type.)

```java
double pi = 3.14159;
System.out.printf("The value of pi is %f", pi);
```

```
The value of pi is 3.14159
```

Flags - %f

- , flag - Inserts commas
- .*N* flag - Rounds to *N* decimal places

```java
double pi = 3.14159;
System.out.printf("The value of pi is %.2f", pi);
```

```
The value of pi is 3.14
```

- Note: comma flag must proceed .*N* flag

Format Specifiers - %c and %C

- Indicates the corresponding parameter is a character (char type.)
  - %c forces the character, if it's a letter, to lowercase.
  - %C forces the character, if it's a letter, to uppercase.

```
char aChar = 'a';
char bChar = 'B';
char cChar = 'c';
char dChar = 'D';
System.out.printf("Characters: %C%c%c%C", aChar, bChar, cChar, dChar);
```

```
Characters: AbcD
```

Format Specifiers - %n

- Forces a new line.
  - Works just like \n

```
int height = 44;
System.out.printf("The%nheight is %d", height);
```

```
The
height is 44
```