

Java Programming Language Fundamentals

Michael C. Hackett

Assistant Professor, Computer Science

Community
College
of Philadelphia

Lecture Topics

- Creating Java Programs
- Primitive Data Types
- Variables
 - Declarations, Initialization, and Assignment
 - Copying values
- Constants
- Comments
- Console Output
- Strings
 - Concatenation and Appending
 - Calling on a String's method
- Scanner objects/Keyboard input
- Escape Sequences
- Formatted Console Output
 - `System.out.printf()`
- Arithmetic Operators
- Narrowing and Widening Data Types
- Mixed Integer Operations
- Math Object
- Parsing Integer/Double values from Strings

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

Creating a Java Program

- While Java source code is written in text like Python, how the source code is arranged in Java is a little different.
- Java is a heavily object-oriented language.
 - We'll get deeper into object oriented programming later.
- Java source code is contained within objects.
 - A ***software object*** is an abstract idea modeled using variables and subroutines.

Creating a Java Program

- The first step in writing a Java program is declare your object using a class header.
- The class header begins with the keyword **class** followed by the name that we want to call the object.
 - The braces { and } contain the ***class body***- All of the code contained in the object.

```
class Example {  
  
}
```

Creating a Java Program

- The source code file's name must match the name of our class.
- The Example object below (and from the previous slide) must be contained in a file named Example.java

```
class Example {  
  
}
```

Creating a Java Program

- A main method is the starting point for a Java application.
 - The main method must be written as shown below.
 - Since everything is contained in objects in Java, all subroutines are technically methods.
 - We'll ignore much of the keywords like "public" and "static" for the time being.

```
class Example {
```

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

```
}
```

- Java does not require indentation like Python does.
 - Indentation does make code neater/more readable.
 - Braces, { and }, are used.

Creating a Java Program

- When the Java application starts, it begins executing the code in the main method.

```
class Example {  
  
    public static void main(String[] args) {  
        System.out.println("Hello World");  
    }  
  
}
```

Hello World

Notable Difference in Syntax

- Executable statements in Java must be punctuated with a semicolon.

```
class Example {  
  
    public static void main(String[] args) {  
        System.out.println("Hello World");  
    }  
  
}
```

The Eight Primitive Data Types in the Java Programming Language

- Integer Types
 - byte, short, int, long
- Floating Point Types
 - float, double
- Character Type
 - char
- Boolean Type
 - boolean

Integer-Type Primitives

- **byte** (8 bits)
 - Can represent any integer between -128 and 127
- **short** (16 bits)
 - Can represent any integer between -32,768 and 32,767
- **int** (32 bits)
 - Can represent any integer between -2,147,483,648 and 2,147,483,647
 - Most frequently used integer primitive.
- **long** (64 bits)
 - Can represent any integer between -2^{63} and $2^{63}-1$

Floating Point-Type Primitives

- **float** (32 bits)
 - Can represent values between $\sim \pm 3.4 \times 10^{38}$ with 7 significant digits.
- **double** (64 bits)
 - Can represent values between $\sim \pm 1.7 \times 10^{308}$ with 15 significant digits.

Character-Type Primitive

- **char** (16 bits)
 - Can represent a single, 16-bit Unicode character.
- UTF-16 character table for reference:
<http://www.fileformat.info/info/charset/UTF-16/list.htm>

Boolean-Type Primitive

- **boolean** (1 bit)
 - Can be **true** (1) or **false** (0).
 - Used for decision making.
- Depending on the OS's memory management, it may not be able to allocate a single bit of memory.
 - The OS may allocate an entire byte (8 bits), though only one of the bits will be used.

Declaring a Variable

- When a variable is ***declared*** in Java, you are stating:

- The type of data this variable will reference.
- The name of the variable.

- Examples:

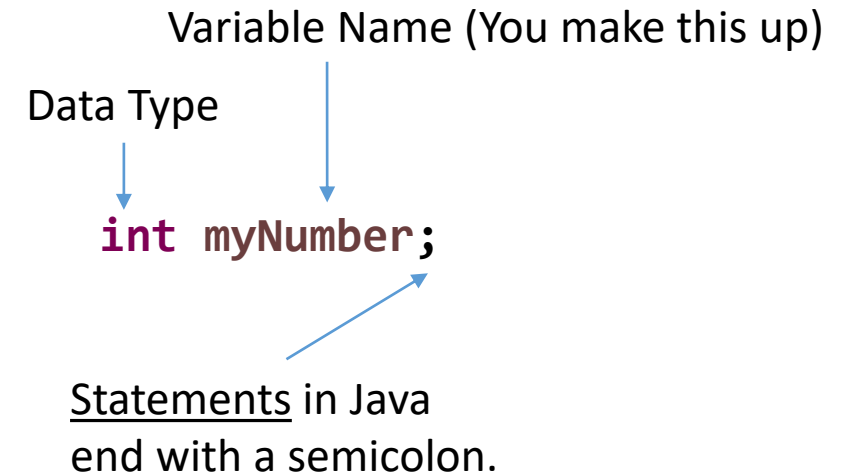
```
int age;  
double temperature;  
char letter;  
boolean ready;
```

Variable Name (You make this up)

Data Type

int myNumber;

Statements in Java
end with a semicolon.



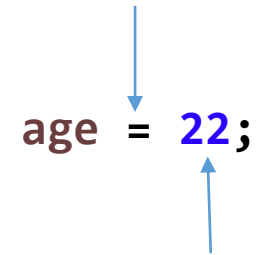
- Variables in Java are ***statically typed***.
 - It cannot reference values of a different data type after it has been declared.

Initializing a Variable

- A variable becomes ***initialized*** when an initial value is assigned to the variable.
- Assignment operator: =
 - Same as Python
- Examples (all previously declared in the last slide):

```
age = 22;  
temperature = 80.5;  
letter = 'A';  
ready = true;
```

Assignment Operator



age = 22;

Initial value

- Note: You cannot initialize a variable that has not yet been declared!

Declare and Initialize a Variable

- You can assign an initial value when you declare a new variable.

- Examples:

```
int age = 22;  
double temperature = 80.5;  
char letter = 'A';  
boolean ready = true;
```

Declaration

```
int age = 22;
```

Initialization

Literals (Integer)

- byte, short and int literals can be expressed in
 - Decimal (Base 10)
 - Octal (Base 8), or
 - Hexadecimal (Base 16)
- Base 10 Literal (No prefix): `int decimalNumber = 100;`
- Base 8 Literal* (0 prefix): `int octalNumber = 0144;`
- Base 16 Literal* (0x prefix): `int hexNumber = 0x64;`

*For the purpose of this course, we will only be using base 10 values.
It's good to know that other numeric literals exist, though.

Literals (Integer)

- long literals (Must add a lowercase or capital L to the end):

```
long exampleLongLiteral = 255L;
```

- A capital L is preferred, as lowercase L may be mistaken for a one.

Literals (Floating Point)

- double literals (Nothing special needs to be done):

```
double exampleDoubleLiteral = 255.23;
```

- float literals (Must add upper- or lowercase f to the end):

```
float myExampleFloat = 15.5f;
```

The compiler (like with many other programming languages/compiler) interprets literal fractional numbers as doubles by default. To differentiate float literals from double literals, float literals must end with a lowercase f.

Literals (Character)

- char literals can be expressed as a character literal, a Unicode literal or a number.
- **Must be in single quotes!!**

```
char exampleCharLiteral = 'A';  
char exampleCharUnicodeLiteral = '\u0041';  
char exampleCharDecimalLiteral = 65;
```

- UTF-16 character table for reference:

<http://www.fileformat.info/info/charset/UTF-16/list.htm>

Any classwork will use character literals like exampleCharLiteral above.

Changing a Variable's Reference

- ***Assignment/Reassignment*** is replacing the existing value referenced by a variable with a new value.
 - The new data must be of the correct type.
- Like initialization, also uses the Assignment Operator, =
- Example:

```
double temperature = 67.5;  
temperature = 68.2;
```

Copying references from one variable to another


- Use the Assignment Operator, =
- Examples:

```
double speed = 35.2;  
double speed2 = speed;
```



Value of speed is copied to speed2

```
int apples = 10;  
int apples2;  
apples2 = apples;
```

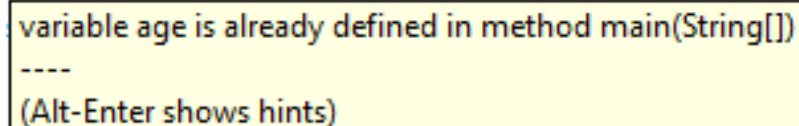


Value of apples is copied to apples2

A Few Notes on Variables in Java

- The default values of the primitive data types (their values before any initialization is performed) are:
 - byte, short, int, long : 0
 - float, double : 0.0
 - char : ''
 - boolean : false
- Variable names must be unique, regardless of data type.

```
int age;  
double age;
```



```
variable age is already defined in method main(String[])  
-----  
(Alt-Enter shows hints)
```

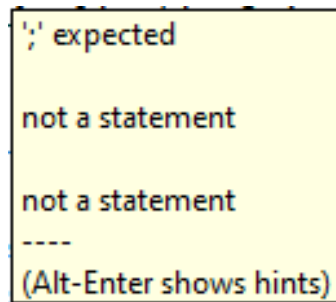
Compile-Time Error will occur.

- A compile-time error occurs when you compile your code.
- A run-time error occurs when your program is running.

A Few Notes on Variables in Java

- A variable's data type cannot be changed after declaration.
- The names of the primitive data types are all Java keywords.
 - Keywords cannot be the name of a variable.

`int boolean;` ← Compile-Time Error.



';' expected
not a statement
not a statement

(Alt-Enter shows hints)

Naming Variables

- Names must start with a letter, dollar sign, or underscore.
- Names may contain numbers, but **cannot** start with numbers.
- Aside from letters, dollar signs, underscores, and numbers, no other characters may be used.
- Names cannot contain spaces. Use underscores, if necessary.

`int someName;` Valid.

`int _someName;` Valid. Can start with underscore.
`int some_Name;` Valid. Can contain any underscores.

`int $someName;` Valid. Can start with dollar sign.
`int some$Name;` Valid. Can contain any dollar signs.

`int 0someName;` INVALID. Can't start with a number.
`int some0Name;` Valid. Can contain any numbers.

`int some Name;` INVALID.
`int some_Name;` Valid.

Variable Names

- Variable names in Java normally start with a lowercase letter.
 - Object names normally start with an uppercase letter.
- “Camel-case” is the convention used for variable names in Java.
 - For variable names that are multiple words long, the first letter of every subsequent word should be capitalized.

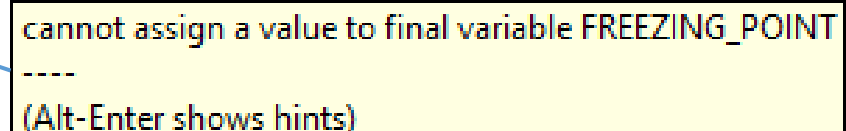
```
bottlesOfBeerOnTheWall = 99  
hasBeenDeleted = True
```

Constants

- A **constant** is a variable whose value cannot be changed.
 - Constants are declared by using the **final** keyword.
 - By convention, the name of a constant is in uppercase with underscores between words.
- Examples:
- Trying to assign a constant a new value will result in a compile time error.

```
final int FREEZING_POINT = 32;  
final double PI = 3.14159;  
final char LETTER_ZEE = 'Z';
```

```
final int FREEZING_POINT = 32;  
FREEZING_POINT = 0;
```



cannot assign a value to final variable FREEZING_POINT

(Alt-Enter shows hints)

Comments

- Single line comments begin with //

```
//Single line comment
```

- Multiple line comments begin with /* and end with */

```
/* Everything between slash-asterisk  
and asterisk-slash  
will be  
ignored*/
```

- Comments are entirely ignored by the compiler. You can type whatever you want in a comment.

Typical Convention for Commenting

```
/**  
 * This program will ask users for input  
 * and then display some output.  
 */  
public static void main(String[] args) {  
    int temp = 10; //Initialize temperature to ten.  
    ...  
}
```

System object

- The System object is provided by Java and allows access to standard input and output.
 - The default output stream is simple text output to the console/terminal.
 - The default input stream is the keyboard (which we will later use to allow a user to type input into a program.)
- The System object is the first object we will be using in the course.
 - We are utilizing the objects and code contained in the System object within our own program.

System.out

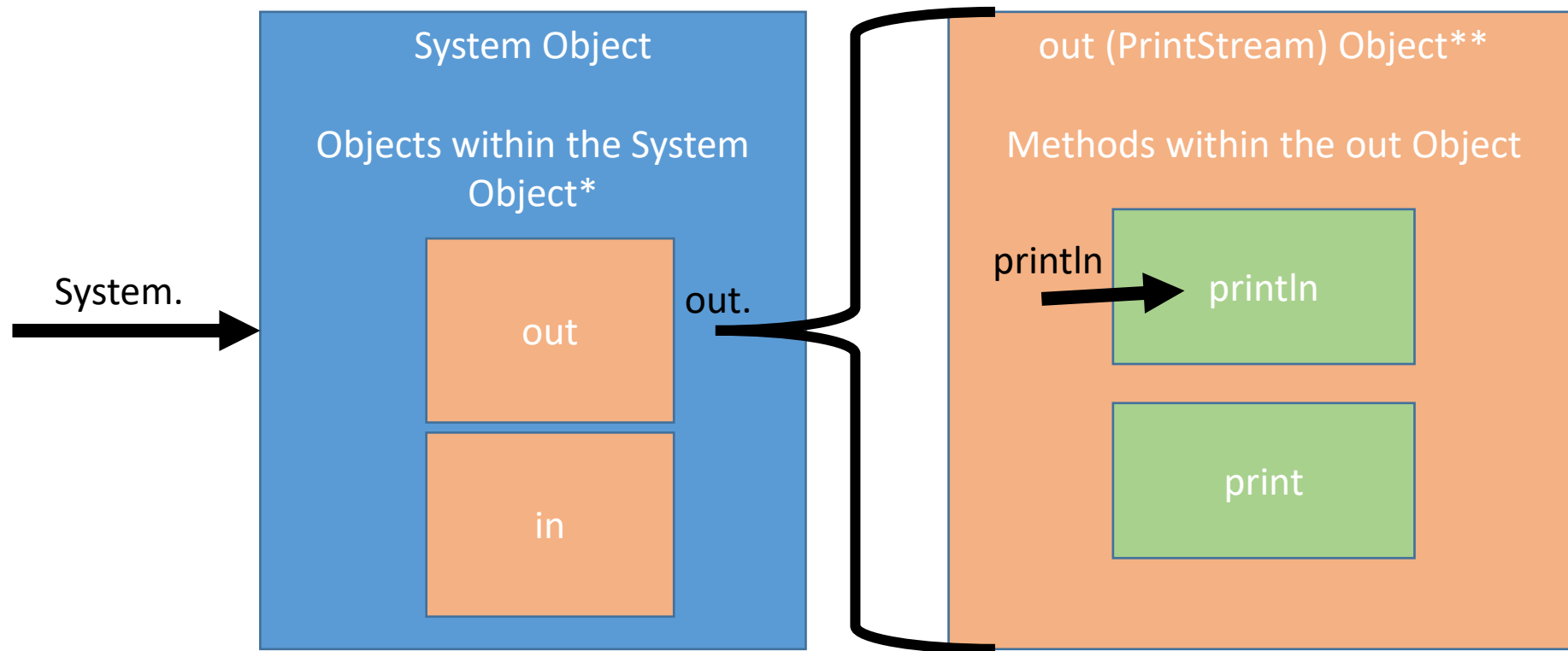
- Inside of the System object, it has a `PrintStream*` object named “out”.
 - This object handles all default output, which is ultimately printed out to the terminal/console.
- To access components within an object, we use ***dot notation***.
 - To access the *System* object’s *out* object: **System.out**

* You don’t have to worry about the technical specifics of a `PrintStream` object.

System.out

- There are two methods (sections of pre-written and reusable code) we will be using from System.out – print and println
- To access an object's methods, we again use dot notation - `System.out.println()`
 - We are basically saying, "In the System object's out object, execute the println method's code."
- The parentheses after the method name is for the parameter list.
 - Any data passed as a parameter to the print or println methods will be printed on the screen.

System.out.println() Walkthrough



* System object contains much more than just those two objects. This is just a generalization.

** A PrintStream contains much more than just those two methods. This is also just a generalization.

Console Output - System.out.println()

- After printing the supplied information, the println method will return to the next line.

```
char gradeLetter1 = 'A';  
char gradeLetter2 = 'B';  
System.out.println(gradeLetter1);  
System.out.println(gradeLetter2);
```

A

B

Console Output - System.out.print()

- Unlike the println method, the **print** method will stay on the same line.

```
char gradeLetter1 = 'A';  
char gradeLetter2 = 'B';  
System.out.print(gradeLetter1);  
System.out.print(gradeLetter2);
```

AB

Console Output

- Both methods only accept **one** argument.
- We can NOT do the following:

```
char gradeLetter1 = 'A';  
char gradeLetter2 = 'B';  
System.out.print(gradeLetter1, gradeLetter2);
```



Compile-Time Error.

Strings

- A ***String*** is an object (not a primitive) that contains a sequence of characters.
- The sequence of characters in a String can include any number of:
 - Letters
 - Numbers
 - Symbols
 - Spaces

Strings

- Since a String is an object, it provides methods/functions that can be called to perform operations on the String's data.
- A String provides many methods that make the manipulation of its data relatively painless.
 - We will only see a few today and see others later in the course.

Declaring a String object

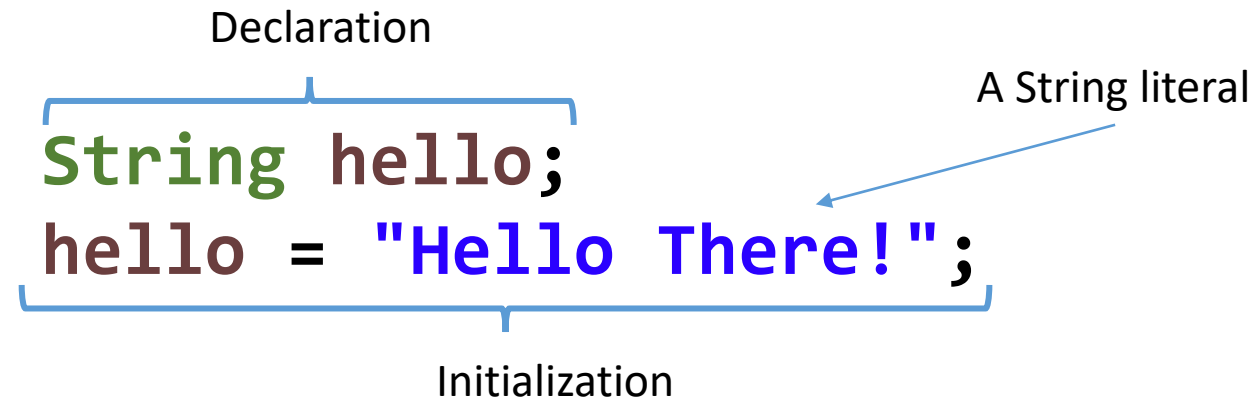
- Strings, like primitives, are declared:
 - First stating the data type.
 - Then stating the variable name.

 **String** hello;

Capital S

Initializing a String object

- Use the assignment operator: =
- A ***String literal*** is any source code representation of a sequence of characters in double quotation marks.



The diagram shows two lines of Go code. The first line is `String hello;` and the second line is `hello = "Hello There!";`. A blue bracket above the first line is labeled "Declaration". A blue bracket below the second line is labeled "Initialization". A blue arrow points from the text "A String literal" to the string `"Hello There!"` in the second line.

```
String hello;  
hello = "Hello There!";
```

Declaration

A String literal

Initialization

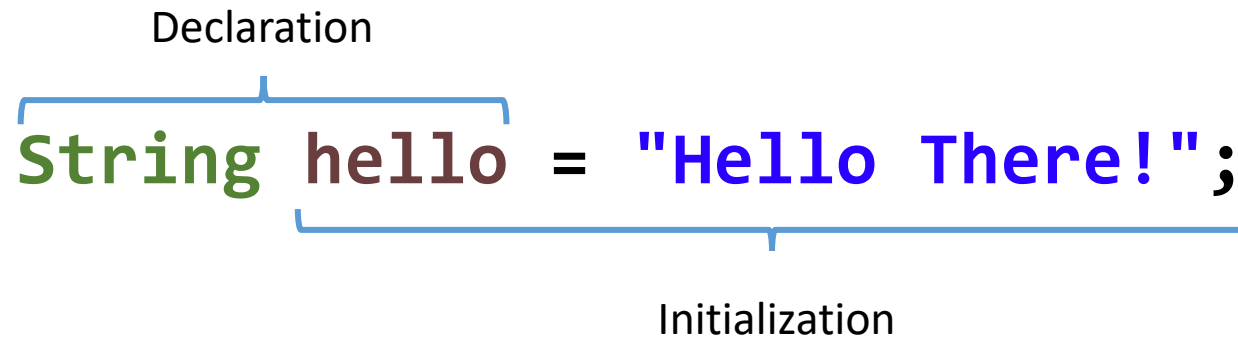
Declare and Initialize a String object

- Declaration and initialization can be done in one statement.

Declaration

```
String hello = "Hello There!";
```

Initialization



Reassigning Strings

- Use the assignment operator: =

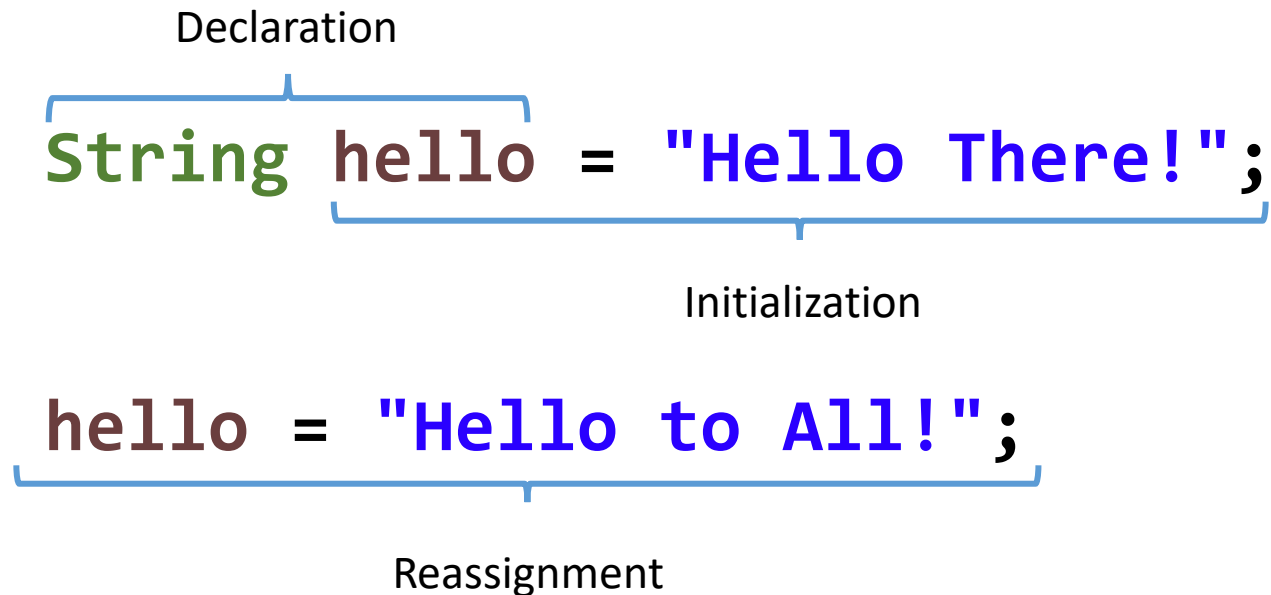
Declaration

```
String hello = "Hello There!";
```

Initialization

```
hello = "Hello to All!";
```

Reassignment

The diagram illustrates three stages of string handling in code. The first line, 'String hello = "Hello There!";', is annotated with a bracket above 'String' labeled 'Declaration' and a bracket below the entire line labeled 'Initialization'. The second line, 'hello = "Hello to All!";', is annotated with a bracket below the entire line labeled 'Reassignment'. The variable 'hello' is consistently colored brown, 'String' is green, and the string literals are blue.

Printing Strings

- Simply pass the String to System.out.print or System.out.println

```
String hello = "Hello World!";  
String niceDay = "Have a nice day!";  
System.out.print(hello);  
System.out.print(niceDay);
```

```
Hello World!Have a nice day!
```

Printing Strings

- Simply pass the String to System.out.print or System.out.println

```
String hello = "Hello World!";  
String niceDay = "Have a nice day!";  
System.out.println(hello);  
System.out.println(niceDay);
```

```
Hello World!  
Have a nice day!
```

String Concatenation

```
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.

```
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.
 - This wasn't possible in Python without converting int or float types to strings.

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

There are 31 days in January

Using Augmented Assignment to Append

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

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

See you in 2 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);
```

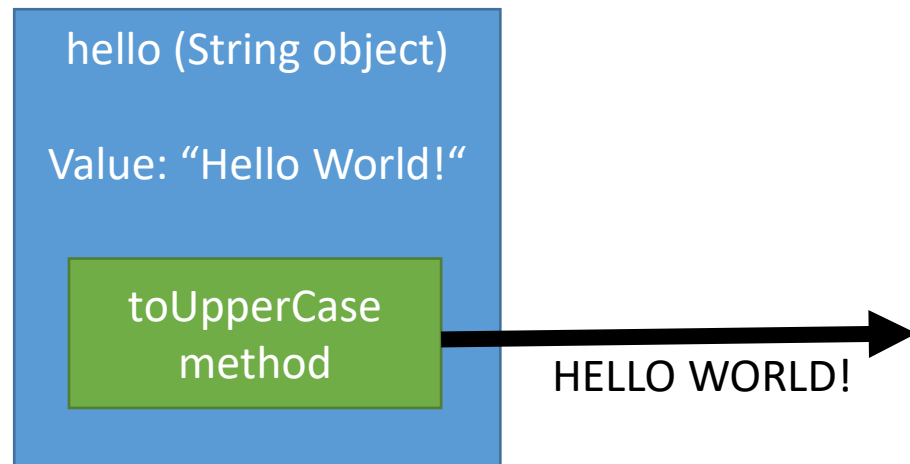
HELLO WORLD!

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?

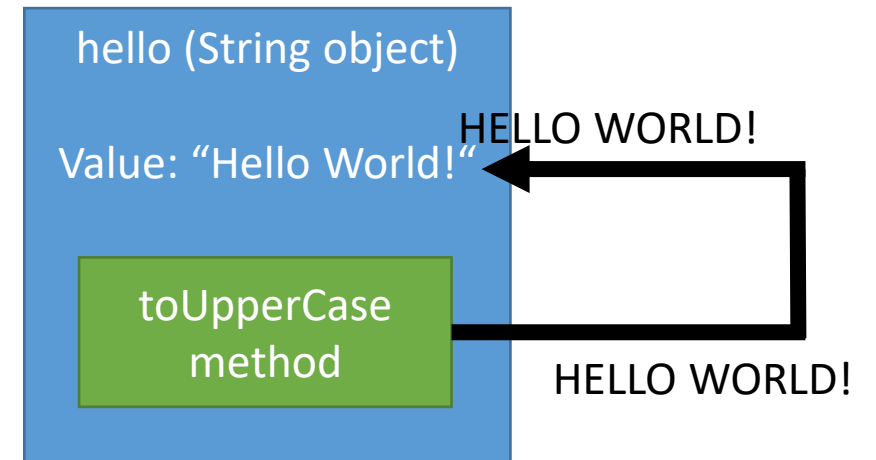
```
String hello = "Hello World!";
```

```
hello.toUpperCase();
```



Doesn't actually change the String's value.

```
hello = hello.toUpperCase();
```



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

```
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.

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

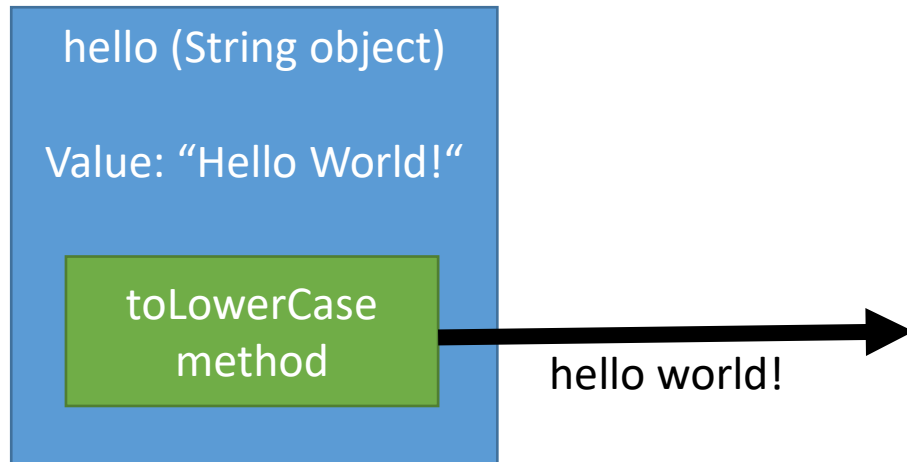
```
hello world!
```

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?

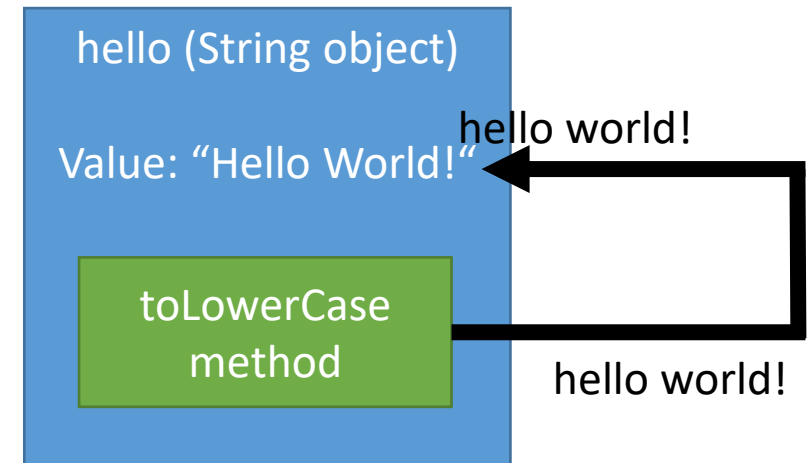
```
String hello = "Hello World!";
```

```
hello.toLowerCase();
```



Doesn't actually change the String's value.

```
hello = hello.toLowerCase();
```



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
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.

Getting Keyboard Input

- A Scanner object provides an easy way to get keyboard entries.
- Scanners will need to be imported.
- Include the following line at the very beginning of your source code, before the class header.

```
import java.util.Scanner;
```


Scanner object

- Unlike other objects we have seen, Scanners will need to be ***instantiated***.
 - When you instantiate, you create a new instance of an object. It then becomes its own entity with its own data.
- Objects are instantiated using the **new** keyword.

```
Scanner keyboard = new Scanner(System.in);
```

When using a Scanner to get keyboard input, I usually name it “keyboard.”
You can name it whatever you like, though.

System.in

- When instantiating a Scanner, you MUST give it an input stream.
- The System object's "in" object is an InputStream object, which a Scanner will accept.
- System.in allows us access to the standard input stream (STDIN), which, in most cases, is the keyboard.

```
Scanner keyboard = new Scanner(System.in);
```

Keyboard Input - Getting a String

```
String someString;  
Scanner keyboard = new Scanner(System.in);  
System.out.print("Enter a String: ");  
someString = keyboard.nextLine();  
System.out.println("The String you entered is: "  
                    + someString);
```

Enter a String: Hello World!

The String you entered is: Hello World!

Scanner object methods*

Method	Return Type	Description	Possible Exceptions**
nextLine()	String	Returns the next line of input in String form.	NoSuchElementException IllegalStateException

*- There are more than just this one method. We will use more as the semester progresses.

** - It is unlikely you will see these exceptions when using System.in as the Scanner's input stream.

Escape Sequences

- `\n` – Line Feed
 - `\t` – Tab
 - `\"` – Double Quote
 - `\\` – Backslash
-
- There are more, but we will only be working with these few.

Escape Sequences - \n

- \n inserts a line feed (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.

```
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.)

```
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 and 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.)

```
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

```
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
```

Arithmetic Operators

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

Addition

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

- The variable *sum* is initialized with the value 11.

Subtraction

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

- The variable *difference* is initialized with the value -1.

Multiplication

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

- The variable *product* is initialized with the value 30.

Division

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

- The variable *quotient* is initialized with the value 2.
- Java only has one division operator.
 - Python has / and //
 - // starts a single-line comment in Java.

Division (Another example)

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

- 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.

Remainder/Mod Division

- Finds the remainder of a division.

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

- The variable *remainder* is initialized with the value 3.
- “11 divided by 4 is 2 with a remainder of 3”

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;
```

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.

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

42

20

22

Augmented 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;
```

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 myDouble;  
int myInt = 34653;  
myDouble = myInt;
```

- 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 myDouble = 453.87;  
int myInt;  
myInt = (int)myDouble;
```



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 myInt is 453 NOT 453.87

Converting Primitives - Narrowing

- If you forget to typecast when narrowing, your code will not compile.

```
double myDouble = 453.87;  
int myInt;  
myInt = myDouble;
```

Compile-time error

incompatible types: possible lossy conversion from double to int

(Alt-Enter shows hints)


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: 1000110010100


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 -108 (Way off from 4500)

Mixed Integer 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 Integer 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

Mixed Integer Operations

```
byte byte1 = 10;
```

```
byte byte2 = 15;
```

```
byte sumOfBytes;
```

```
sumOfBytes = byte1 + byte2;
```

incompatible types: possible lossy conversion from int to byte

(Alt-Enter shows hints)

- 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**!

Mixed Integer Operations

- This will work:

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



Results in an int

Mixed Integer Operations


- You can also narrow the result to the desired type:

```
byte byte1 = 100;  
byte byte2 = 100;  
byte sumOfBytes;  
sumOfBytes = (byte)(byte1 + byte2);
```

- However, you need to be careful about overflow.

Mixed Integer Operations

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



incompatible types: possible lossy conversion from int to short

(Alt-Enter shows hints)

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

Math object

- Provides access to mathematical functions like rounding, exponents, and square roots.
- Unlike Python's math module, Java's Math object does not need to be imported.
 - It's always available to call on.

Math object – Rounding

- Round method takes a **double or float**.
 - Returns a **long** if double.
 - Returns an **int** if float.

```
double originalNumber;  
long roundedNumber;  
originalNumber = 25.6;  
roundedNumber = Math.round(originalNumber);  
System.out.println("The rounded number is: " + roundedNumber);
```

The rounded number is: 26.0

Math object – Rounding Up

- Ceiling (ceil) method takes a **double or float**.
 - Always returns a **double**.

```
double originalNumber;  
double roundedNumber;  
originalNumber = 15.1;  
roundedNumber = Math.ceil(originalNumber);  
System.out.println("The rounded number is: " + roundedNumber);
```

The rounded number is: 16.0

Math object – Rounding Down

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

```
double originalNumber;  
double roundedNumber;  
originalNumber = 18.8;  
roundedNumber = Math.floor(originalNumber);  
System.out.println("The rounded number is: " + roundedNumber);
```

The rounded number is: 18.0

Math object – Square Root

- Square Root function takes any numeric value;
 - Always returns a **double**.

```
int myNumber;  
double squareRoot;  
myNumber = 16;  
squareRoot = Math.sqrt(myNumber);  
System.out.println("The square root is: " + squareRoot);
```

The square root is: 4.0

Math object – Exponents

- Power (pow) function takes two parameters: base and exponent
 - Always 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

Math object methods

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

Parsing Integers from Strings

```
String ten = "10";  
int result = ten + 15;
```

incompatible types: String cannot be converted to int

(Alt-Enter shows hints)

- The above code will not work! You cannot perform arithmetic with Strings, even if the String's characters are numbers.
 - The result of `ten + 15` would actually be `"1015"` not `25`. Java would treat this as concatenation since one of the operands is a String.
- 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.

```
int ten = "10";
```

incompatible types: String cannot be converted to int

(Alt-Enter shows hints)

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

Integer object

- The Integer object is a special type of object known as a “wrapper” object.
 - 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

```
String ten = "10";  
int result = Integer.parseInt(ten) + 15;  
System.out.println("The result is " + result);
```

The result is 25

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

```
String tenFive = "10.5";  
double result = Double.parseDouble(tenFive) + 15;  
System.out.println("The result is " + result);
```

The result is 25.5

NumberFormatException

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

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

run:

```
[-] Exception in thread "main" java.lang.NumberFormatException: For input string: "abcd"  
    at sun.misc.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:2043)  
    at sun.misc.FloatingDecimal.parseDouble(FloatingDecimal.java:110)  
    at java.lang.Double.parseDouble(Double.java:538)  
    at javaapplication3.JavaApplication3.main(JavaApplication3.java:20)  
Java Result: 1
```

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.

Keyboard Input - Getting numeric values

- Scanners have `nextInt()` and `nextDouble()` methods for getting int values and double values that are typed in.
- I recommend not using them; They get the value, but they ignore the user pressing the enter key. This causes problems the next time you ask a user to enter some data.
- Get numeric values as Strings first, then parse them out as ints or doubles.

Keyboard Input - Getting numeric values

```
String someString;  
int someInt;  
Scanner keyboard = new Scanner(System.in);  
System.out.print("Enter a String: ");  
someString = keyboard.nextLine();  
System.out.print("Enter a Number: ");  
someInt = Integer.parseInt(keyboard.nextLine());  
System.out.println("The String you entered is: "  
                    + someString);  
System.out.println("The Number you entered is: "  
                    + someInt);
```

Keyboard Input - Getting numeric values

```
Enter a String: Hello World!
```

```
Enter a Number: 15
```

```
The String you entered is: Hello World!
```

```
The Number you entered is: 15
```