I. Methods

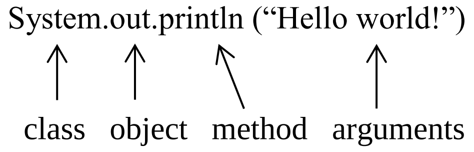
week 4

- a method consists of a sequence of instructions that can access the internal data of an object (**strict** method) or to perform a task with input from arguments (**static** method)

- when you call the method, you **do not** have to know exactly what those instructions are, or even how the object is organized internally

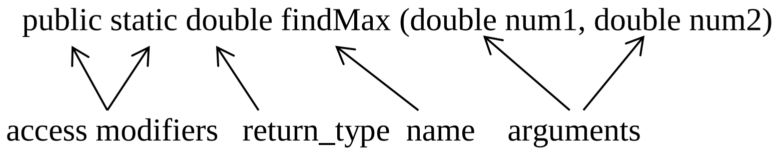
- however, the behavior of the method is well-defined, and that is what matters to us when we use it

- built-in methods:



System.out.println (“Hello world!”) ; // note this is an invocation

- user-defined methods:



public static double findMax (double num1, double num2) // definition

- printf method

  System.out.printf (“string to output %format_specifier”, object_to_format);

  - example format specifier: %8.2f
  - System.out.printf (“The answers are %8.2f and %6.2f ”, num1, num2);
  - note the multiple variations of printf, this is an example of **operator overloading,** that is one operator/command performing multiple actions based upon the context in which it is used, also known as its signature

- see the .getCurrencyInstance method page on syllabus

- see programming assignment1

II. Miscellaneous

- line continuation

  - ➢ since java uses statement terminators, a line can be broken in most any white space

- escape characters

  - ➢ a character preceded by the backslash character \ which has special meaning to the compiler

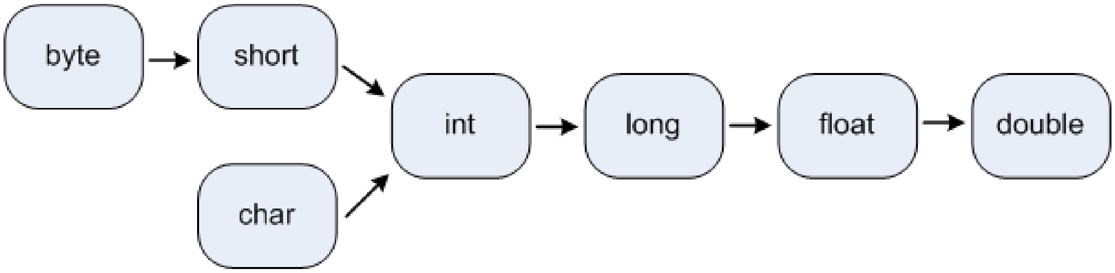
  - ➢ common escape chars include \n (newline), \t (tab character), \char (ignore any special meaning of char), e.g. \\

III. Type Casting

- converts (casts) a primitive data type to another primitive data type

1. **implicit**, aka widening or up-casting

   - ➢ converts a smaller primitive type to a larger primitive type



   - ➢ examples:

```
double d = 123;
int i = ‘a’;                // single quotes make it a char type
```

2. **explicit**, aka narrowing or down-casting

- converts a larger primitive type to a smaller primitive type

- note this results in loss of data!

- syntax: new_small_value = (small_type) big_value;

- examples:

```
int i = (int) 34.567;          // results in 34

int i = 34.567;                // error


float xFloat = 3.45678f;        // literal causes number to be a float type
float sum;

sum = 2.0 + xFloat;             // error, why
```

this will not work because, by default, in Java the 2.0 on the right-hand-side of the last expression is a double-precision number, so "xFloat" is first converted to a double-precision number before it is added to "2.0", then right-hand-side is of type "double" while the left-hand-side is of type "float" so you have to perform an explicit type cast, like....

```
sum = (float) (2.0 + xFloat);     or     sum = 2.0f + xFloat;
```

IV. Conversion Methods (pg 326)

- Java’s conversion methods convert data to/from a String type to another primitive type

- if numeric data is input as a String type, it must be converted to a numeric type to perform calculations

- conversion methods are static (not strict) methods, that is they operate only on arguments

- recall methods return values across the assignment operator (=) in 1 direction only, L ← R

- conversion types must match (see below)

- syntax: class_variable = Class.method (“string_arg”);

- <u>from String examples</u>

  int i = Integer.parseInt (“1234”);

  double d = Double.parseDouble (“12.34”);

  int i = Integer.parseInt (“12.34”); // error

- <u>to String examples</u>

  String s = Integer.toString (123);

V. Getting Input Continued

- using Scanner .nextInt or .nextDouble works for simple valid input, but not very well for anything else

- to verify our input, the safe way is to 1) read all input as a String type, then 2) (try to) convert the String type to the type we wish, using the conversion methods discussed above

- we will use the Scanner method .nextLine to read a String of text from the console

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

String strNumber ;
double number;

strNumber= in.nextLine();                          // step 1
number = Double.parseDouble (strNumber);           // step 2

    // OR, combining 2 steps into a single line

number = Double.parseDouble (in.nextLine());       // don't need strNumber
```

VI. Math Class methods **(pg 882)**

- used to calculate more complex math functions

- types are usually double, and number and types of arguments depend upon specific method

- general format: answer = Math.method_name (argument(s)) ;

  examples:

```
double d = Math.abs (x);          // returns the absolute value of x

double d = Math.pow (x, y) ;      // returns the value of x raised to the y power
```

- note that to perform a simple exponentiation such as x ^2, it is much simpler and faster to do

```
double d = x * x;          rather than          double d = Math.pow (x, 2);
```

- see java for argument types as well as return types