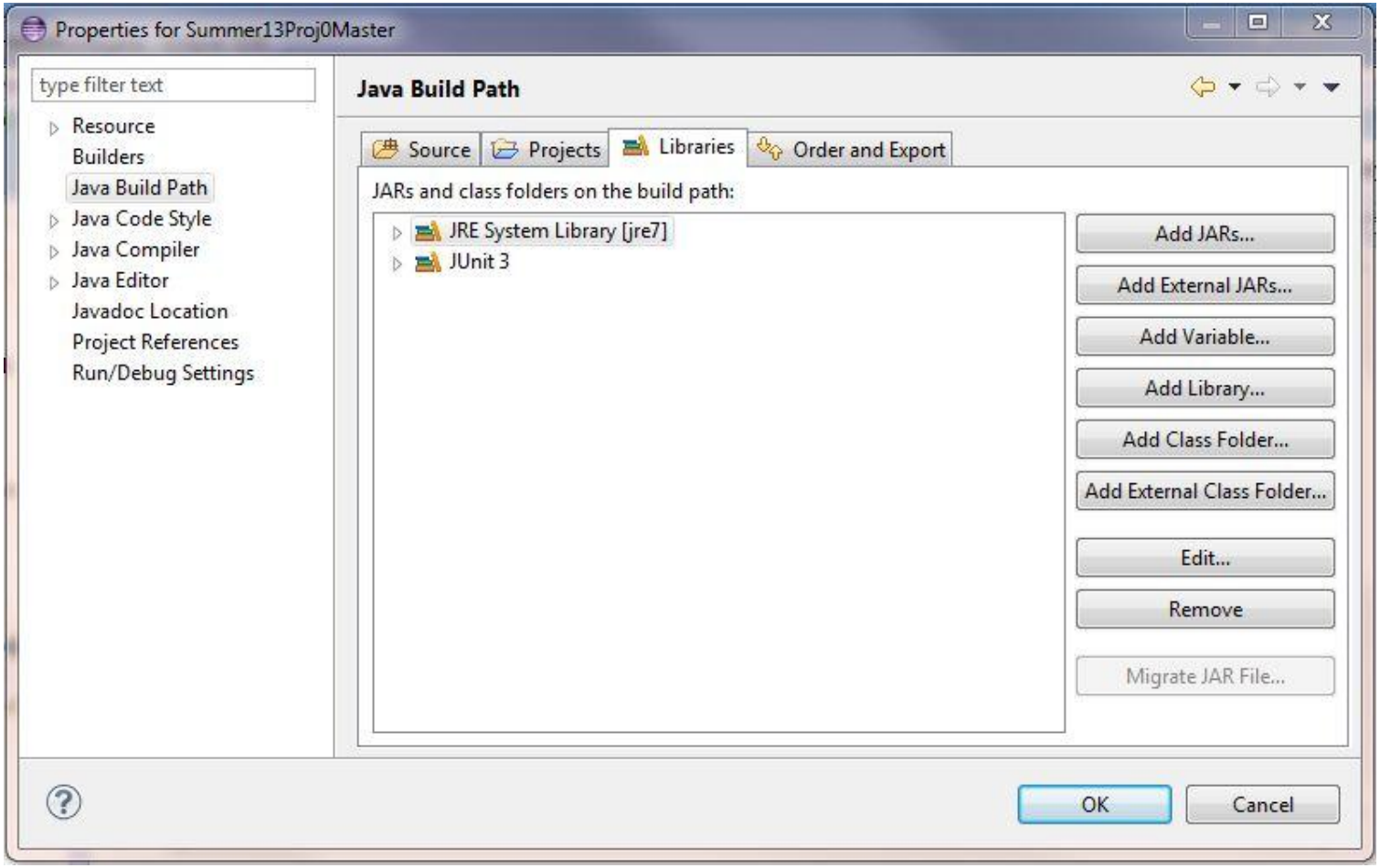
Announcements

- Quiz Thursday
  - Covers everything through this Wednesday: Architectures and software, primitive types, variables, operators, expressions, control flow
  - Practice questions will be posted on the webpage by the end of the day
- Lab tomorrow
  - Submission is resolved on the server side, make sure things work on your end

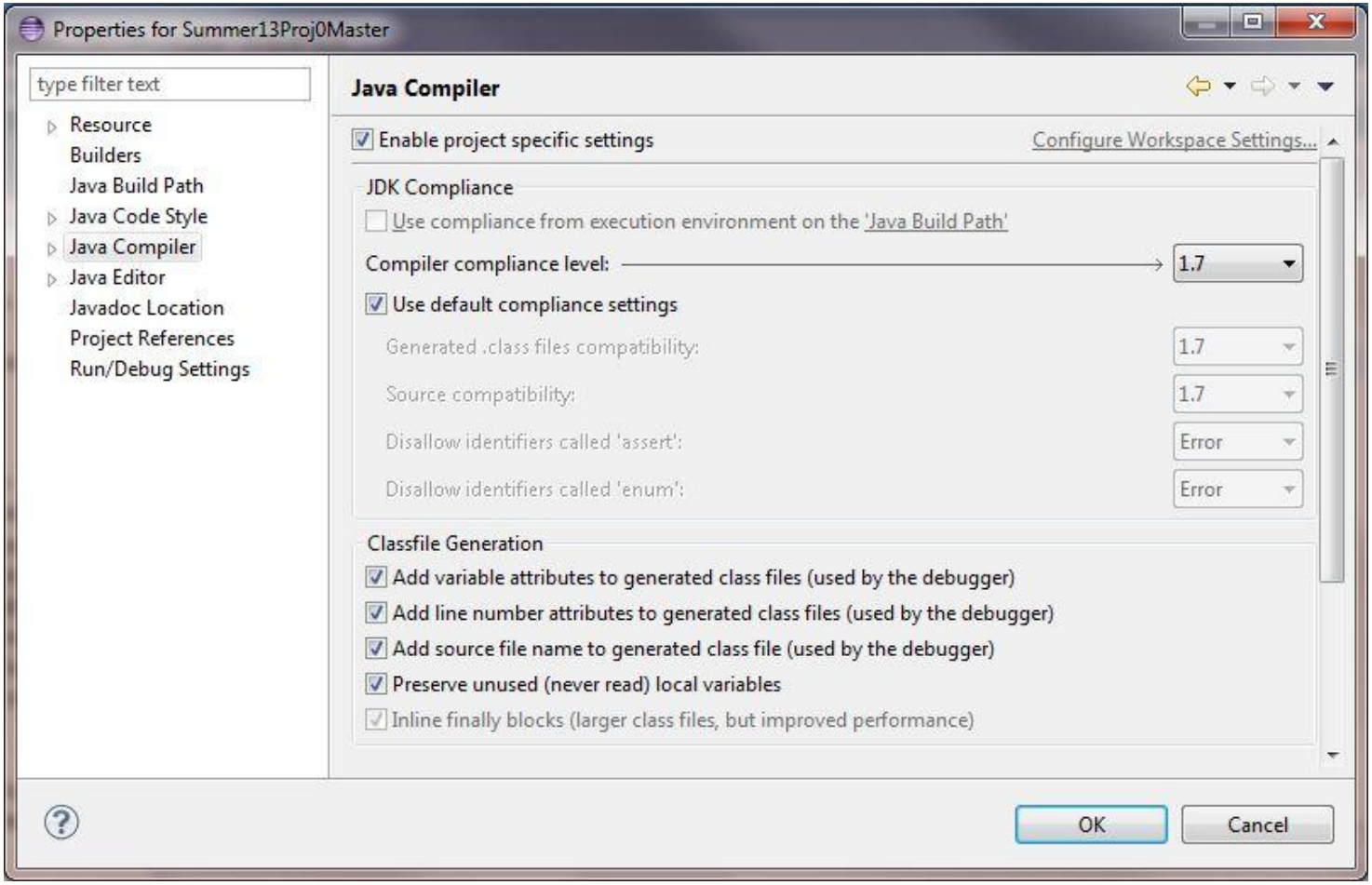
Trouble shooting Magic line errors

- Project -> properties -> Build Path -> Add Library/Remove
- Make sure you are using JRE7 if you have it, otherwise try JRE6



Trouble shooting “class loader” exceptions

- Project -> properties -> Compiler-> JDK Compliance
- Try reducing to 1.6



Last resorts

- Manually download the lab.java file from the schedule on the class webpage
- Manually upload your completed lab.java file to the submit server using the web form at

  submit.cs.umd.edu/summer2013

Project 0

- This will be posted as soon as the submit server is setup for automated testing
- Automated test results on submit.cs.umd.edu/summer2013:

Submissions

#	submitted	public tests	release tests		submission		Public								Release						
			score	when	Details	Download	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7
3	Mon, 08 Apr at 10:01 PM	50 / 50	20 / 20		view	download															
2	Sun, 07 Apr at 06:43 PM	42 / 50	20 / 20		view	download															
1	Sun, 07 Apr at 04:41 PM	26 / 50	18 / 20		view	download															

Expressions and Evaluation

What is an Expression?

- 3 -> ??
- 3+5 -> ??
- (3+5)*10 -> ??
- (3+5)*10 > 7 -> ??
- (3+5)*10 > factorial(7) -> ??
- ((3+5)*10 > 7) && (myBool1 || myBool2)

What is an Expression?

- 3 -> 3
- 3+5 -> 8
- (3+5)*10 -> 80
- (3+5)*10 > 7 -> true
- (3+5)*10 > factorial(7) -> false
- ((3+5)*10 > 7) && (myBool1 || myBool2)

What is an Expression?

- Evaluates to a single value
- Can contain operators, methods, literals, variables, and other expressions
- Can be combined indefinitely to produce more complex structure
- Not a command – it has a value, but it doesn't do anything.

What is an Expression?

"An *expression* is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language, that evaluates to a single value."

http://docs.oracle.com/javase/tutorial/java/nutsandbolts/expressions.html

Evaluating Expressions

Evaluate sub-expressions left to right, deep to shallow:

```
( 5 + (x=3) ) + ( ( 4 * (x++) ) + x )
```

x	2

Evaluating Expressions

Evaluate sub-expressions left to right, deep to shallow:

```
( 5 + (x=3) ) + ( ( 4 * (x++) ) + x )
(     ?     ) + (         ?         )
```

x 2

Evaluating Expressions

Evaluate sub-expressions left to right, deep to shallow:

```
( 5 + (x=3) ) + ( ( 4 * (x++) ) + x )
(     ?     ) + (          ?        )
  5 + ( ? )
```

x | 2 |

Evaluating Expressions

Evaluate sub-expressions left to right, deep to shallow:

```
( 5 + (x=3) ) + ( ( 4 * (x++) ) + x )
(     ?     ) + (          ?        )
  5 + ( ? )
      x=3
```

x 2

Evaluating Expressions

Evaluate sub-expressions left to right, deep to shallow:

```
( 5 + (x=3) ) + ( ( 4 * (x++) ) + x )
(     ?     ) + (          ?        )
  5 + ( 3 )
      x=3
```

x | 3 |

Evaluating Expressions

Evaluate sub-expressions left to right, deep to shallow:

```
( 5 + (x=3) ) + ( ( 4 * (x++) ) + x )
(     8     ) + (          ?        )
  5 + ( 3 )
      x=3
```

x | 3 |

Evaluating Expressions

Evaluate sub-expressions left to right, deep to shallow:

```
( 5 + (x=3) ) + ( ( 4 * (x++) ) + x )
(     8     ) + (         ?         )
  5 + ( 3 )           (     ?     ) + ?
      x=3
```

x | 3 |

Evaluating Expressions

Evaluate sub-expressions left to right, deep to shallow:

```
( 5 + (x=3) ) + ( ( 4 * (x++) ) + x )
(     8     ) + (       ?         )
  5 + ( 3 )           (     ?     ) + ?
      x=3               4 * ( ? )
```

x | 3 |

Evaluating Expressions

Evaluate sub-expressions left to right, deep to shallow:

```
( 5 + (x=3) ) + ( ( 4 * (x++) ) + x )
(     8     ) + (       ?         )
  5 + ( 3 )           (     ?     ) + ?
      x=3               4 * ( ? )
                          x++
```

x | 3 |

Evaluating Expressions

Evaluate sub-expressions left to right, deep to shallow:

```
( 5 + (x=3) ) + ( ( 4 * (x++) ) + x )
(     8     ) + (       ?         )
  5 + ( 3 )           ( ? ) + ?
      x=3           4 * ( 3 )
                        x++
```

x 4

Evaluating Expressions

Evaluate sub-expressions left to right, deep to shallow:

```
( 5 + (x=3) ) + ( ( 4 * (x++) ) + x )
(     8     ) + (       ?           )
  5 + ( 3 )           ( 12 ) + ?
      x=3           4 * ( 3 )
                        x++
```

x 4

Evaluating Expressions

Evaluate sub-expressions left to right, deep to shallow:

```
( 5 + (x=3) ) + ( ( 4 * (x++) ) + x )
(     8     ) + (        ?          )
  5 + ( 3 )           ( 12 ) + 4
      x=3           4 * ( 3 )
                        x++
```

x 4

Evaluating Expressions

Evaluate sub-expressions left to right, deep to shallow:

```
( 5 + (x=3) ) + ( ( 4 * (x++) ) + x )
(     8     ) + (       16          )
  5 + ( 3 )           ( 12 ) + 4
      x=3           4 * ( 3 )
                        x++
```

x 4

Control Flow

If statements, switches, and methods

One more operator

The ternary operator:

A ? B : C

"If A is true, evaluate to B, else evaluate to C"

- This is an expression, not a command.
- The inputs are also expressions, not commands.

If-else statements

More versatile, human-readable version of ternary operator:

```
if(A) { //value of A is boolean
      /* Execute some commands (not
       * expressions)
       */
} else {
      /* Execute some other commands
       * instead
       */
}
```

If-else variants

Each { … } is called a “block.”

The else block is optional:

```
if(A) {
    //Do something
}
//otherwise, do nothing
```

If-else variants

Multiple cases are allowed:

```
if(A) {
    //Do something
} else if(B) {
    //Do something else
} else if(C) {
    //Yet a different something
} else {
    //Do this if A,B,C were all false
}
```

Non-determinism

- Multiple possible outcomes for a single program.
- Control flow is most interesting in non-deterministic programs.
- Sources of non-determinism:
  - Random number generation

```
// A random double in (0.0, 1.0):
double myRand = Math.random();
```

  - User-Input
    - Java uses a “scanner” object to read input from the console

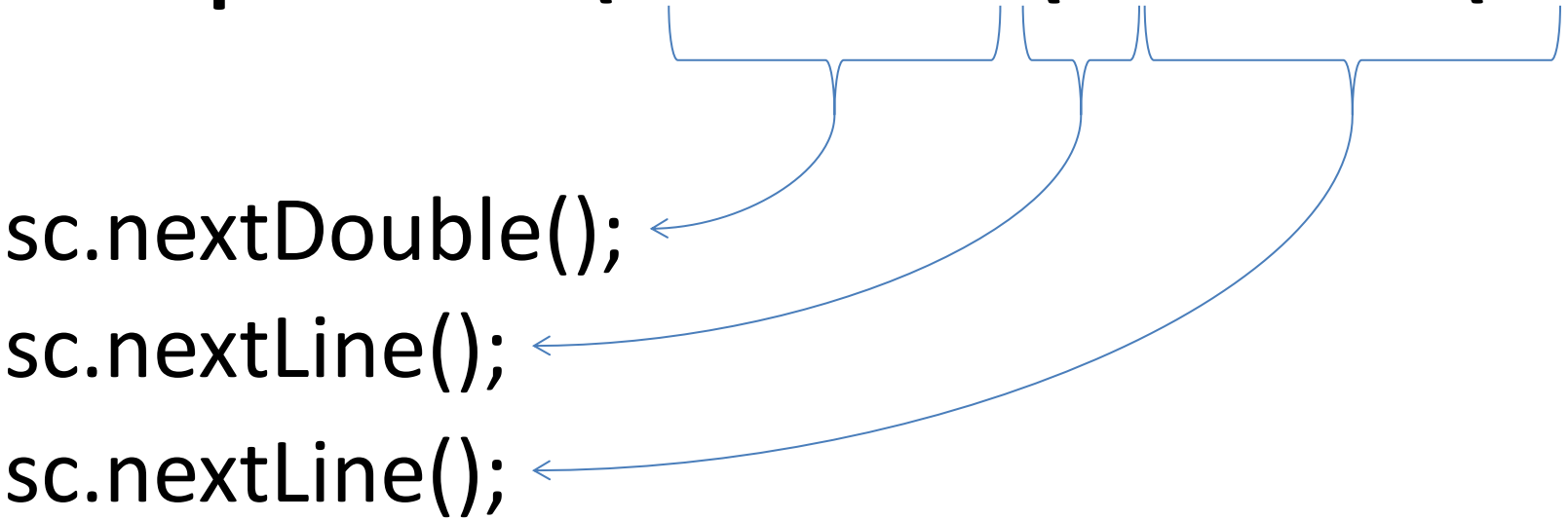
Using a scanner

```
//Get a new scanner for reading input
Scanner sc = new Scanner(System.in);

//Read a double from input (also has nextInt, etc)
double myDouble = sc.nextDouble();

//Read a line of input (including carriage return)
String myLine = sc.nextLine();

//Close the scanner when done
sc.close();
```

Scanners and carriage returns

- Methods like nextDouble() do not consume a trailing carriage return:

Input: “\n0.123\nHello\n”



Switch statements

Switch statements are an alternative way of handling many possible cases:

```
Switch(myInt)  {
       case 2:
              //Do this if myInt==2
              break;
       case 5:
              //Do this if myInt==5
              break;
       default:
              //Otherwise do this
              break;
}
```

Switch statements

- Only the following types are allowed as switch variables (e.g. myInt):
  - byte, short, char, int (primitives)
  - Byte, Short, Character, Integer (Objects, more later)
  - Enumerated types (more later)
  - Strings (In Java 7)

Switch statements

- Any case labels can be used as long as they are constants of the correct type
  - Literals are constants
  - Variables declared with the “final” keyword are constants:

```
final int z = 0; //z can't be changed
```

- Execution starts at the first matching case label, and proceeds until it encounters a `break` statement
- If there is no matching case label, execution starts at the `default` label.

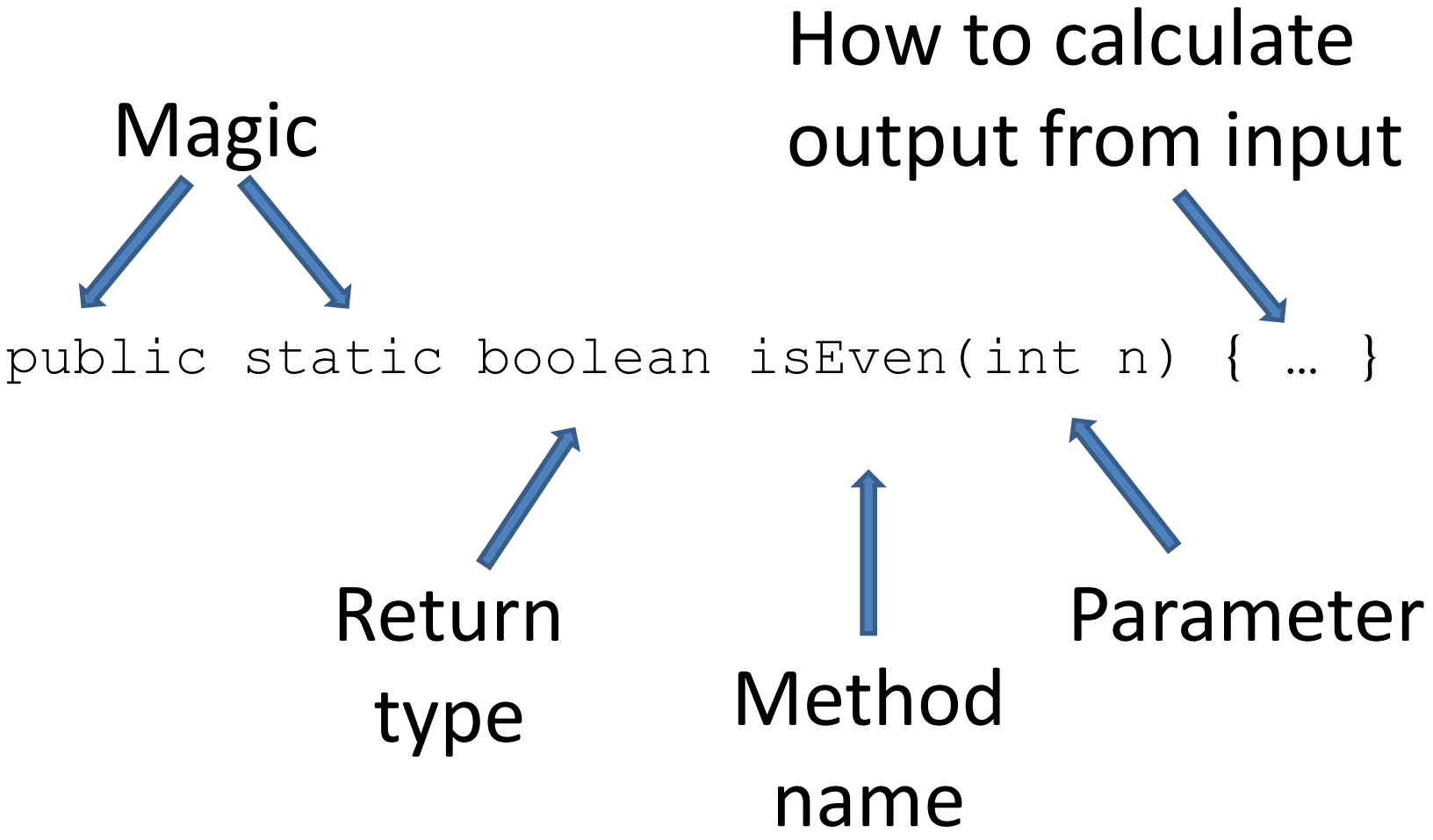
(public static) methods

A method is an operator that the programmer defines. Java needs to know:

- The name of the method
- What are the inputs (called *parameters*)
- What type of output (called *return type*)
- How to calculate the output from the input

"Public" and "Static" are object-oriented concepts (coming up next week).

(public static) methods



Returning output

- When a method is done calculating, it can *return* an output value, using the `return` keyword.
- A method need not return anything. In this case the return type is `void`.
- The `return` keyword can be used anywhere inside the method; once it is reached, subsequent statements are ignored.

Example method

```
public static boolean isEven(int n) {

      //Calculate output from input
      if(n % 2 == 0) { //Check input
            return true; //Return output
      } else {
            return false; //Return output
      }

}
```

Flow of execution

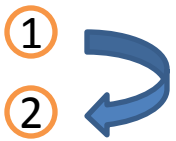
```
public static void main(String[] args) {
        int a = 6;
        boolean b = isEven(a);
        System.out.println(b);
}

public static boolean isEven(int n) {

        if(n % 2 == 0) {
                return true;
        } else {
                return false;
        }

}
```

1

a

Flow of execution

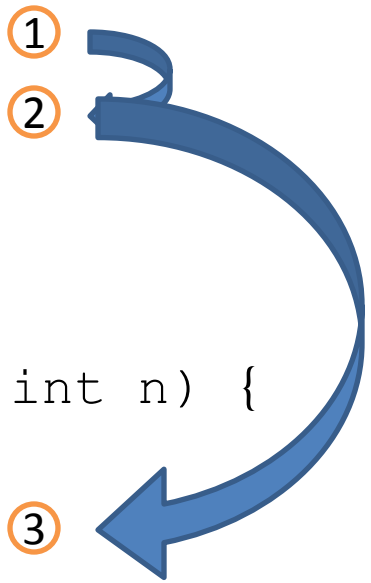
```
public static void main(String[] args) {
        int a = 6;
        boolean b = isEven(a);
        System.out.println(b);
}

public static boolean isEven(int n) {

        if(n % 2 == 0) {
                return true;
        } else {
                return false;
        }

}
```

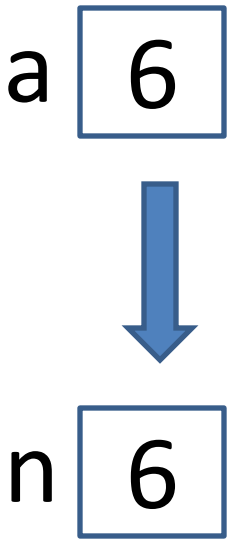


a | 6

Flow of execution

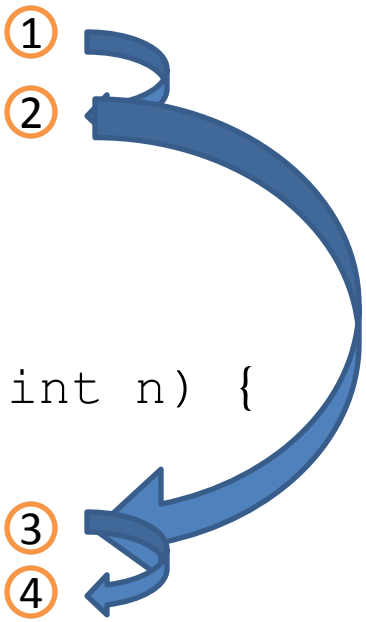
```
public static void main(String[] args) {
        int a = 6;
        boolean b = isEven(a);
        System.out.println(b);
}

public static boolean isEven(int n) {

        if(n % 2 == 0) {
                return true;
        } else {
                return false;
        }

}
```

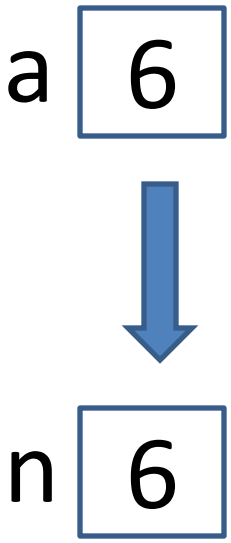




Flow of execution

```
public static void main(String[] args) {
        int a = 6;
        boolean b = isEven(a);
        System.out.println(b);
}

public static boolean isEven(int n) {

        if(n % 2 == 0) {
                return true;
        } else {
                return false;
        }

}
```





Flow of execution

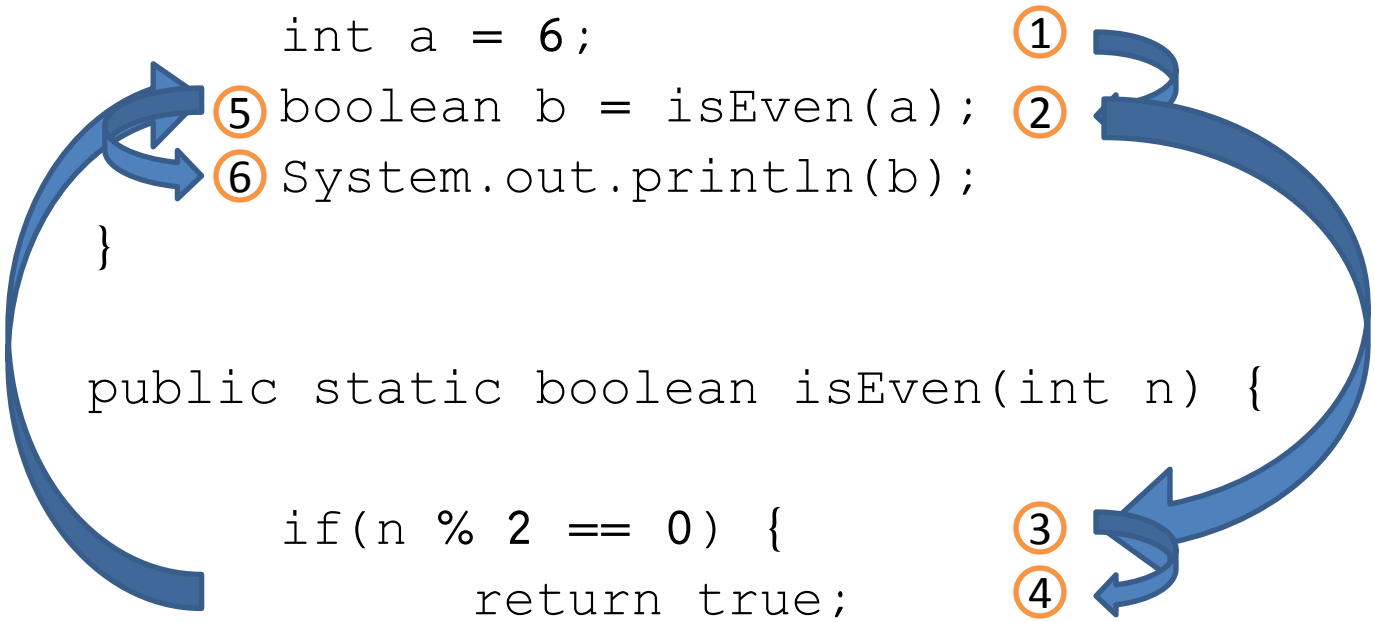
```
public static void main(String[] args) {
        int a = 6;
        boolean b = true;
        System.out.println(b);
}

public static boolean isEven(int n) {

        if(n % 2 == 0) {
                return true;
        } else {
                return false;
        }

}
```

1
2
3
4
5

a 6

Flow of execution

```
public static void main(String[] args) {
        int a = 6;
        boolean b = isEven(a);
        System.out.println(b);
}

public static boolean isEven(int n) {

        if(n % 2 == 0) {
                return true;
        } else {
                return false;
        }

}
```

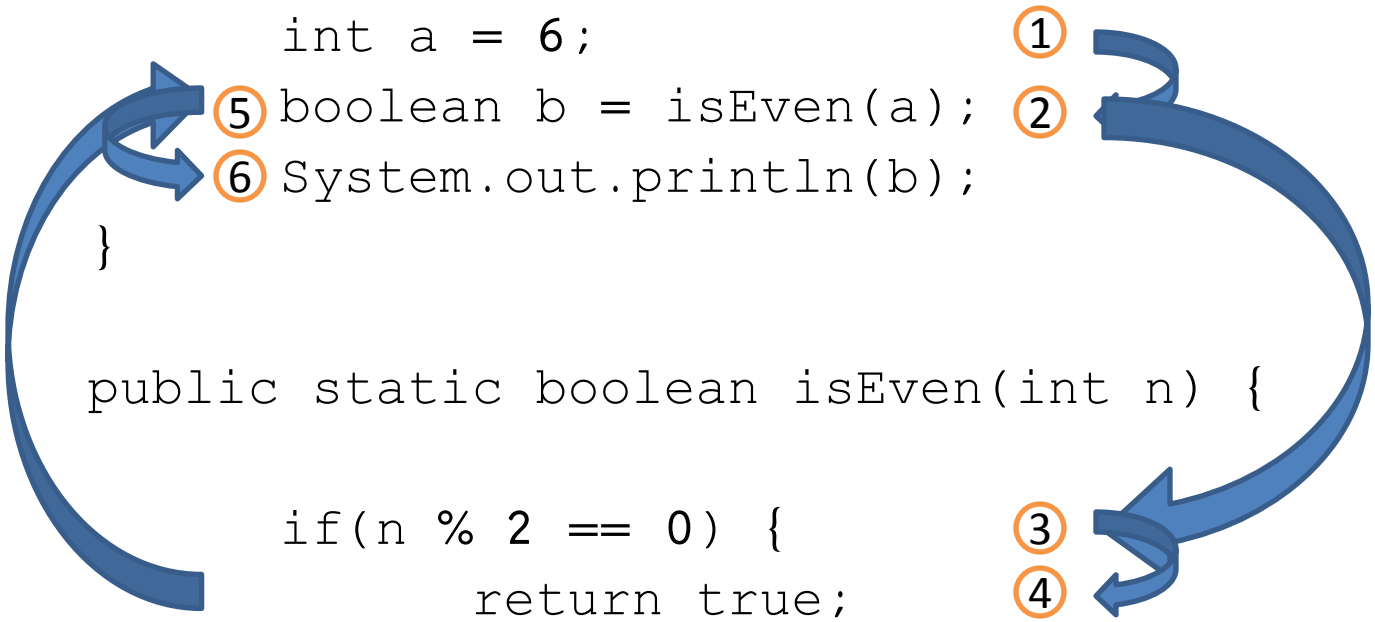


a 6

b T

Flow of execution

```
public static void main(String[] args) {
        int a = 6;
        boolean b = isEven(a);
        System.out.println(b);
}

public static boolean isEven(int n) {

        if(n % 2 == 0) {
                return true;
        } else {
                return false;
        }

}
```



a | 6

b | T

"True"

Parameters and arguments

- Parameters are the input variables specified in the method definition
- Arguments are the input values *passed* to the method when it is *called*
- When the method executes, parameters contain ***copies*** of the original arguments.
  - Changes to the parameters do not affect the original arguments.
- Args and Params example

Local Scope

- All variables declared in a method definition (including parameters) have *local scope*.
  - They are inaccessible from outside the method
  - Their values do not persist from one call to the next
- To make things confusing, local method variables can have the same name as variables elsewhere in the code.
  - The external variable is completely independent from the local one
  - This is called *shadowing*